Consistent Software Cities

Supporting Comprehension of Evolving Software Systems

Von der Fakultät für Mathematik, Naturwissenschaften und Informatik
der Brandenburgischen Technischen Universität Cottbus

zur Erlangung des akademischen Grades

Doktor der Naturwissenschaften
(Dr. rer. nat)

genehmigte Dissertation

vorgelegt von

Dipl.-Inf.
Frank Steinbrückner
geboren am 09.11.1977 in Eisenhüttenstadt

Gutachter:	Prof. Dr. Claus Lewerentz
Gutachter:	Prof. Dr. Douglas W. Cunningham
Gutachter:	Prof. Dr. Rainer Koschke

Tag der mündlichen Prüfung: 27. November 2012

Abstract

Software visualization is a field of software engineering which aims at providing visual representations of software systems or particular aspects thereof. Numerous approaches for the visualization of software systems have been developed during the last decades. Software cities denote one particular kind of software visualizations that adopts the city metaphor for depicting software systems as virtual cities. Due to their high expressiveness and effectiveness software cities are mainly used for program comprehension tasks, during reverse engineering, and in quality analysis scenarios.

Software systems evolve. They are steadily enhanced to provide new product functions, corrected to fix defects, or adapted to changed system environments. Software cities, as proposed today, do not take this evolutionary character of software systems into account. The evolution of software systems, i.e. changes to their internal structure and other characteristics, can easily disrupt the overall software city structure and thus yield very inconsistent visualizations for evolving software systems. The interpretation of these evolving software cities may be error-prone and time-consuming.

In this thesis we propose a new software city approach that is based on the observation that real cities often reveal their evolution insofar as they contain e.g. historic centers or satellite cities. Analogous patterns, however, cannot be found in software cities which is an astonishing divergence. We claim that by preserving historical structures and carefully expanding software cities during software evolution, i.e. by spatializing software evolution, we obtain highly expressive, effective, and consistent software cities which support a broader range of application scenarios than the current state of the field.

The main contribution of this thesis is a new layout approach for software cities, which explicitly takes evolution into account. Its effects are twofold: First, evolution becomes directly visible in the software city structure in the form of specific geographic patterns that each depicts particular evolutionary phenomena. The resulting increased expressiveness allows for supporting new application scenarios which are evaluated for several example systems. Second, software evolution does no longer disrupt the overall software city structure. Instead, the software cities evolve smoothly during system evolution, which allows for using them during ongoing system development and maintenance. The high consistency of this approach is confirmed in an empirical evaluation.

Zusammenfassung

In den vergangenen Jahren wurden zahlreiche, teils sehr unterschiedliche Ansätze für die Visualisierung von Softwaresystemen vorgestellt. Softwarestädte sind ein spezieller Visualisierungsansatz, in dem Strukturen und Eigenschaften von Softwaresystemen in Form virtueller Städte dargestellt werden können. Aufgrund ihrer hohen Ausdruckskraft und Effektivität werden Softwarestädte hauptsächlich zur Programmanalyse, insbesondere während Reverse-Engineering Aktivitäten und in der Softwarequalitätsanalyse eingesetzt.

Ein wesentliches Problem bisheriger Ansätze liegt in der Instabilität von Softwarestädten gegenüber Softwareänderungen. Selbst kleinste Änderungen an Softwaresystemen können schnell zu gravierenden strukturellen Änderungen der sie repräsentierenden Softwarestädte führen. Die dabei entstehenden Inkonsistenzen der Visualisierungen verschiedener Versionsstände erschweren die Interpretierbarkeit, insbesondere die Nachverfolgbarkeit von Softwareänderungen, und verstärken somit das Risiko von Fehlinterpretationen. Ein entwicklungsbegleitender Einsatz von Softwarestädten mit dem Ziel der kontinuierlichen Überwachung und frühzeitigen Erkennung von Entwicklungsproblemen wird somit erschwert.

Während sich in realen Städten im Verlauf ihrer Entwicklung typische Strukturen, wie z.B. historische Stadtkerne oder Vorstädte, herausbilden, lassen sich analoge Muster in bisherigen Softwarestädten nicht finden. Gerade auch in Softwarestädten lassen sich durch die Erhaltung historischer Strukturen jedoch ausdrucksstarke, effektive und vor allem konsistente Visualisierungen formen. In dieser Arbeit wird daher ein neuer Ansatz für die Visualisierung von Softwaresystemen als Softwarestädte vorgestellt, in dem die Entwicklungshistorie von Softwaresystemen berücksichtigt, die Strukturen in Softwarestädten erhalten und ihr entwicklungsbegleitender Einsatz somit ermöglicht wird.

Der Hauptbeitrag der Arbeit ist ein neuer Ansatz für die Visualisierung dynamischer hierarchischer Strukturen. Angewandt auf Softwarestädte hat dieser Ansatz gegenüber bisherigen Ansätzen zwei wesentliche Vorteile: Zum einen wird Evolution direkt sichtbar in Form bekannter städtischer Strukturen, die jeweils ein spezifisches Entwicklungsmuster repräsentieren. Wie im Verlauf der Arbeit gezeigt wird, ermöglicht die sich daraus ergebende erhöhte Ausdruckskraft der Visualisierungen die Unterstützung neuer Analyseszenarien. Zum anderen zerstören Softwareänderungen nicht mehr die grundlegende Struktur der Softwarestädte, wodurch insbesondere ihr entwicklungsbegleitender Einsatz mit dem Ziel der kontinuierlichen Projektüberwachung ermöglicht wird.

Acknowledgements

This research was performed at the Software Systems Engineering Research Group at the Brandenburg University of Technology Cottbus. I want to express my special thanks to my supervisor Prof. Dr. Claus Lewerentz for offering me the opportunity to work on this topic and for contributing many ideas to this thesis.

I want to express special thanks to Markus Uhlig who has put an enormous amount of work into the development and continuous improvement of the CrocoCosmos tool.

The volume of this thesis would not have been possible without the collaboration of many students who directly and indirectly contributed both ideas and solutions to this dissertation in the context of their diploma, master, or bachelor theses. I want to express my special thanks to Martin Neubert for many fruitful discussions that finally led to the EVOSTREETS approach, Michael Tauer [174] for his high quality implementation and evaluation of the early EVOSTREETS approaches, Uwe Mannl [132] for an initial analysis of layout consistency of several layout approaches, Martin Everth [67] for his concepts and discussion of software city landscapes, Martin Junghans [96] and Marcus Uhlig [182] for their concepts of routing edges in fixed graph layouts, as well as Marcel Zierenberg [207], Katharina Legde [121] and Markus Uhlig [183] for the visualization of runtime data, test data, and concepts in software cities. Also, I want to express my thanks to all those students who contributed and evaluated additional ideas that were not directly included into this dissertation.

Furthermore, I want to thank Marcel Bennicke, Mathias Radicke, and Thomas Noack for a very pleasing collaboration during the development of the SOFTWARE COCKPIT.

*

Thank you, Claudia. For reading and for listening.

Contents

List of Figures

List of Tables

There are only two mistakes one can
make along the road to truth;
not going all the way, and not starting.

Buddha

Chapter 1

Introduction

From early requirements elicitation to long term maintenance activities software development is constrained by functional, quality, organizational, economic, and social requirements, and involves a wide range of both customer and production side participants. Software systems, on the other hand, are large and complex systems that are developed and maintained over long periods of time. The trend towards even larger and more complex systems continues and conflicts with strong economic restrictions and requirements concerning software quality.

With the invention of new software engineering processes, methods, and technologies, and the support of huge amounts of engineering tools, software engineering as a discipline has successfully addressed many of the problems arising from an ever increasing software size and complexity over the last decades. But despite this progress, software engineering is still handicapped by an intrinsic characteristic that clearly distinguishes computer programs from most industrial products: Software is an inherently intellectual artifact. It is invisible, immaterial, or as [10] state:

> “Software is intangible, having no physical shape or size. After it is written, code "disappears" into files kept on disks.”

Software visualization is a software engineering field which aims at providing visual representations of software systems or particular aspects thereof. Goals that are often associated with visualization, particularly software visualization, are to amplify cognition ([39]), gain understanding and insight into data ([63]), facilitate human understanding ([149]), to reduce complexity of the phenomena under consideration ([100]), improved productivity ([57]) and savings in time ([20]), better comprehension ([20], [147], [57]) and reasoning ([147]).

Koschke [105] conducted a survey in which researchers in software maintenance, reverse engineering, and re-engineering assessed the role of software visualization for software engineering: More than 80% rate software visualization as *important but not critical* (42%) or *absolutely necessary* (40%) for software maintenance, re-

verse engineering, and re-engineering. Thus, from a research point of view, software visualization is an accepted and established field in software engineering.

1.1 Problem Description

During the last decades an enormous amount of different software visualization approaches ranging from simple sourcecode prettyprinters to sophisticated visualization metaphors has been proposed ([57], [176], [40]). In this thesis, we focus on one particular kind of software visualization, i.e. the visualization of software systems as virtual cities. Software cities are based on the metaphor of real cities. They typically consist of entities like city districts, buildings, infrastructure like streets, and many more to depict software structures and particular characteristics of software elements. Software cities are no new idea. Several different approaches have already been proposed, and tools (from academia and industry) are available. We discuss these approaches in detail in chapter 2.

Current research in software cities shows two important benefits of visualizing software systems as cities:

- Expressiveness
  Expressiveness refers to the ability of visualizations to express a particular set of data. The expressiveness of visualizations relies on the availability of a sufficiently large number of graphical entities and corresponding entity properties which can be used for encoding data. As discussed in e.g. [145] and [2], software cities are potentially very expressive visualizations as they offer a wide range of visual entities and corresponding properties.

- Effectiveness
  Whereas expressiveness refers to *what* is visualized, effectiveness refers to *how* it is visualized. Effective visualizations allow for faster interpretation, convey more distinctions, and lead to fewer errors than other visualizations. As discussed in [197], software cities are an effective means of comprehension of large software systems.

Expressiveness and effectiveness are discussed in further detail in chapter 2.1.1. Both are important qualities of software cities. We explicitly address them throughout this thesis.

Due to their expressiveness and effectiveness, software cities allow for an efficient comprehension of various aspects of large software systems. And though, industrial software development only very slowly adopts software cities as an additional means of engineering. While the reasons for this lack of adoption may be manifold, in this thesis we address one particular conceptual problem of software cities that is still widely unaddressed in current software (city) visualization research, i.e. their applicability during ongoing software development and maintenance. In addition to expressiveness and effectiveness, an application during ongoing development requires another highly important criterion to be addressed, i.e. the consistency of visualizations of evolving data.

During exploration users typically gain familiarity with the visualizations. Changes in the data imply changes in the visualization such that users have to regain familiarity by recovering familiar structures, or by identifying adaptations in the visualization. If the data that is visualized evolves, the goal of information visualization designers must be to minimize the efforts necessary to perform tasks like these and to minimize risks of misinterpretation. If not appropriately taken into account, the visualization may suffer from inconsistencies between successive versions, and applying them may become time-consuming and error-prone.

In this work we use the term *consistency* to refer to the quality of visualizations to support continuous understanding of evolving data sets, a quality that is also known as *preserving the mental map* ([139], [33], [157], [151], [95]), *maintaining the mental model* [120], or *resilience to change* ([204]). Visualization consistency is an important but so far neglected quality criterion for software city visualizations. If not appropriately taken into account, the visualization may suffer from inconsistencies between successive versions which cause increased efforts for regaining familiarity by recovering familiar structures, or for identifying changes in the visualization. In general, the interpretation of inconsistent visualizations is more error-prone and more time-consuming.

Software systems evolve. They are steadily enhanced to provide new product functions, corrected to fix defects, or adapted to changed system environments. Software cities, as proposed today, do not take this evolutionary aspect into account. Instead, they mainly support program comprehension and analysis scenarios and thus address the expressiveness and effectiveness criteria discussed above. The evolution of software systems, i.e. changes to their internal structure or other characteristics, can easily disrupt the overall city structure and thus yield very inconsistent visualizations for evolving software systems. Consistency is, however, a crucial premise for using software cities during ongoing software development and maintenance, particularly for monitoring scenarios.

PROBLEM STATEMENT

> Current approaches for visualizing software systems as cities fail as a **consistent** means of comprehension of **evolving** software systems.

1.2 Thesis Statement

The solution to the problem stated above that is proposed in this thesis is based on the observation that real cities often reveal their evolution insofar as they contain e.g. historic centers or satellite cities. Analogous patterns, however, cannot be found in software cities which is an astonishing divergence particularly with regard to the attention that evolutionary analyses received over the last years. We claim that by preserving historical structures and carefully expanding software cities during software evolution, i.e. by spatializing evolution, we obtain highly expressive, effective, and consistent software cities which support a much broader range of application scenarios compared to the current state of the field.

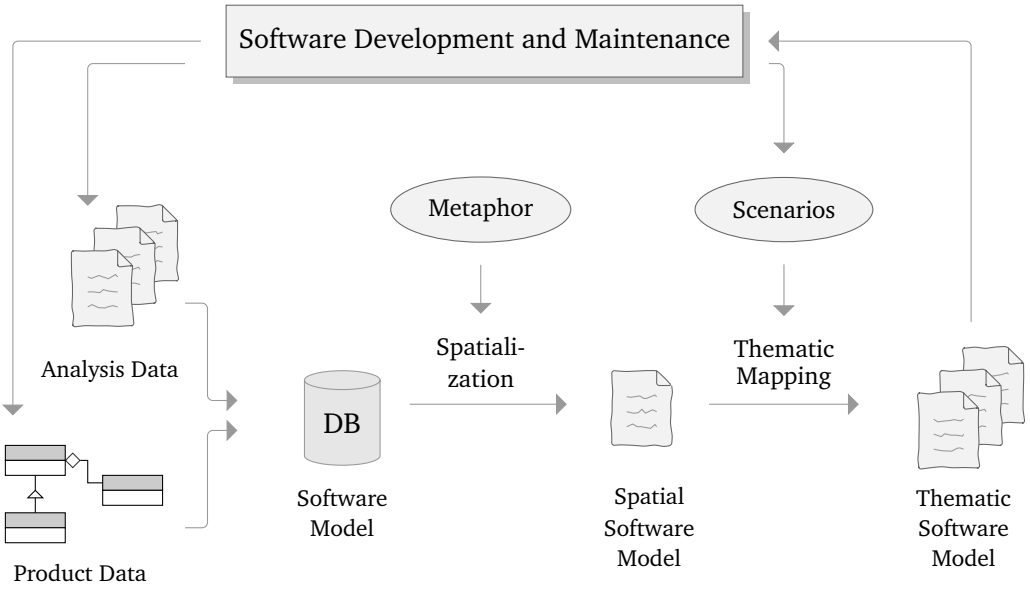


Figure 1.1: Visualization Pipeline for Consistent Software Cities

THESIS STATEMENT

> Properly designed Software Cities are a consistent means of comprehension of large evolving software systems.

To prove this thesis statement we present a new software visualization approach, called *Consistent Software Cities*, that supports various fields of software engineering including program comprehension and reverse engineering, quality analysis and assessment, but also the continuous monitoring of evolving software systems during ongoing software development and maintenance.

1.3 Consistent Software Cities

A clearly defined three-staged visualization pipeline (depicted in figure 1.1) is used for the construction of consistent software cities. It includes the following stages and responsibilities:

- *Software Models* are models of software systems that capture their structure, software analysis data, and the systems' evolution over time. Software models are incrementally populated (e.g. during nightly builds) by an automated analysis tool, the SOFTWARE COCKPIT ([24]), which allows for easily integrating result data from a wide range of different software analysis tools. All models discussed below (and all visualizations in this thesis) are derived exclusively from data stored in the software model.

- *Spatial Software Models* are *spatializations* of software models, more precisely they are software models extended by spatial data. Spatial models are not

visualizations, they cannot directly be visualized but define a spatial arrangement of software elements. Their primary purpose is to provide a stable and expressive spatial reference model for the derivation of thematic software city visualizations.

- *Thematic Software Models* are derived from spatial software models by integrating "thematic" (i.e. analysis) data. The results of this thematic mapping step are application specific thematic *models* of software systems which can directly be visualized as maps using an interactive visualization tool. These thematic software models and their corresponding visualizations emphasize particular aspects of the underlying software system, e.g. code ownership, modification frequencies, test coverage, and the like.

Clearly, similar visualization pipelines are used by other approaches as well. The specific characteristic of our approach is the explicit construction of spatial software models that serve as stable platforms during ongoing system development and from which all thematic software models are derived.

The visualization pipeline is fully tool supported and can directly be integrated into standard development processes.

1.4 Contributions

Software visualization research has already traveled a noticeable part of the road towards expressive and effective software cities (e.g. [146], [2], [31], [195]). The main focus, however, has been put on reverse engineering, program comprehension, and analysis scenarios thereby disregarding another important aspect, i.e. the consistency of software city visualization for evolving software systems. In this work, we shed light on this particular, widely unaddressed aspect and thus contribute one more step towards expressive, effective, and consistent software cities.

This thesis is the first that discusses the consistency of software cities, a property that even though it is such a crucial premise for application during software development and maintenance, has not sufficiently been addressed in previous research. The main contributions of this thesis are as follows:

- From a systematic analysis of currently used layout approaches for software cities, we derive a new layout approach called EVOSTREETS. EVOSTREETS based software cities express the evolution of software systems in the city layout. Software evolution shapes an expressive landscape like city structure which forms geographic patterns that directly depict particular phenomena of software evolution ([173]).

- A systematic empirical evaluation clearly proofs the expressiveness, effectiveness, and consistency of software city visualizations based on the EVOSTREETS approach. This evaluation also includes further layout approaches that are used in other software cities and shows their specific characteristics with respect to the above three quality criteria.

- We provide new thematic mappings which in conjunction with the specific EVOSTREETS layouts reveal new insights into the structure, evolution, and social organization of software projects.
- We further increase the expressiveness of software city visualizations by simultaneously representing hierarchical, evolutionary, and relational data in software city landscapes which also provides us with consistent views between the architecture and design level.

1.5 Structure of this Thesis

Chapter 2 discusses some important topics of the broader field of visualization. Since software systems are immaterial, special emphasis is put on *spatialization* which subsumes fundamental techniques for visualizing non-spatial data. A brief introduction to the particular field of software visualization is included, and a review of the state of the field of visualizing software systems spatially as maps, cities, and landscapes is provided.

Chapter 3 describes the first stage of our visualization pipeline, i.e. the software models for evolving software cities. Software models are in essence dynamic graphs that capture the structure, properties, and evolution of software systems. Also we discuss several relevant topics regarding software structures, software evolution, and aspects of populating the model. Additionally, we present a tool, the SOFTWARE COCKPIT, that allows for steadily and automatically extracting and integrating data from several software analysis tools during system evolution.

Chapter 4 addresses the second stage of the visualization pipeline. First, layout approaches which are currently used for software city visualizations are reviewed and discussed with respect to the three quality criteria expressiveness, effectiveness, and consistency. From this discussion, we derive a new layout approach called EVOSTREETS which explicitly takes development history into account. We illustrate how system evolution becomes directly visible in the form of visual patterns which significantly increases the expressiveness of EVOSTREETS based software city visualizations.

In **chapter 5** the EVOSTREETS approach and other currently used approaches are evaluated empirically with respect to layout consistency and layout effectiveness. We discuss several measures to determine the consistency and effectiveness of these approaches. For this purpose, both characteristics are refined further. The evaluation includes several software systems. It shows clearly that the EVOSTREETS approach produces highly consistent layouts. Its effectiveness, however, suffers from lower layout compactness.

Chapter 6 addresses the third stage of the visualization pipeline. It discusses the construction of thematic software cities on the basis of the EVOSTREETS approach. Thematic software cities encode scenario specific software analysis data. Due to their higher expressiveness and consistency, the EVOSTREETS

based visualizations allow for supporting new application scenarios in the field of program comprehension and reverse engineering, quality analysis and assessment, as well as monitoring of software during evolution. We evaluate several thematic software cities that support these areas of application for a number of academic, industry, and open source software systems.

Chapter 7 presents an additional technique for simultaneously visualizing the coarse-grained system components and their fine-grained design in software city landscapes. The basic idea of software city landscapes is to place several cities that each represents a particular architectural component of the software system in one coherent landscape. Using standard dimensional reduction techniques this placement may additionally reveal the overall coarse-grained dependency structure between software components. Software city landscapes thus can be used to simultaneously depict software systems on a coarse-grained architectural and a fine-grained design level.

Chapter 8 summarizes the main results of thesis and provides an outlook for further research and development.

Visual artifacts and computers do for the mind what cars do for the feet.

Card, Mackinlay, Shneiderman

Chapter 2

Software Visualization

Visualizations are a fundamental technique for human understanding and reasoning. They are applied in nearly all fields of daily life, science, engineering, medicine, the finance sector and so on. Many of the techniques found in software visualization originate from other fields of visualization research. In this chapter, we discuss some important topics of the broader field of information visualization first. Special emphasis is put on *spatialization* which subsumes fundamental techniques for visualizing non-spatial data. A brief introduction to software visualization follows before the state of the field of visualizing software systems spatially as maps, cities, and landscapes is described.

2.1 Information Visualization

The area of visualization can roughly be divided into the subfields information visualization and scientific visualization. Card et al. ([39]) provide definitions for both:

> Information Visualization: The use of computer-supported, interactive, visual representations of abstract data to amplify cognition.
>
> Scientific Visualization: The use of computer-supported, interactive, visual representations of scientific data, typically physically based, to amplify cognition.

While these definitions appear restrictive when requiring computer support and interactivity, the distinction between abstract data and scientific, typically physically based data is commonly accepted (see [170], [57], and [46] for similar definitions). Thus, heat maps of motor engines and medical images of human bodies are examples of scientific visualizations, whereas stock market charts are examples of information visualization. As the use of *typically* indicates, there is no sharp border between scientific and information visualization. Instead both fields intersect [57].

Since computer programs are purely conceptual systems, software visualization is commonly regarded as a branch of Information Visualization ([39], [57]). Many of the techniques found in software visualization originate from information visualization research.

In the remainder of this section we first discuss important quality criteria for information visualization in general, address the question of how to visualize non-spatial data afterwards, and finally discuss differences between the standard visualization pipeline described in the literature and the visualization pipeline that is used in this thesis (see chapter 1).

2.1.1 Design Space and Quality Criteria

Visualizations map data to visual structures which are, as Card et al. [39] discuss, composed of a spatial substrate, marks, and graphical properties of marks. Marks are visual entities that have spatial and retinal properties. Spatial properties denote the position of an entity with respect to the spatial substrate, whereas retinal properties include e.g. color, brightness, or transparency. According to the authors (p.23), two quality criteria, i.e. expressiveness and effectiveness, are highly important for the mapping of data to visual structures. They are defined as follows:

- Expressiveness: "A mapping is said to be expressive if all and only the data in the Data Table are also represented in the Visual Structure."
- Effectiveness: "A mapping is said to be more effective if it is faster to interpret, can convey more distinctions, or leads to fewer errors than some other mapping."

A visualization[1] is not expressive if, for example, the number of visual parameters is not sufficient to express all the data. Effective visualizations allow readers to extract the encoded information efficiently and precisely, i.e. with fewer errors than other visualizations. While expressiveness refers to *what* is visualized effectiveness refers to *how* it is visualized.

Expressive and effective visualizations are not easy to design. Their quality strongly depends on the right mapping of data to visual properties. Comprehensive discussions on how to create high quality visualizations can be found in e.g. [25] and [180]. A more pragmatic ranking of graphical variables with respect to the type of data being visualized (i.e. quantitative, ordinal, nominal) is given by Mackinlay ([129]). Position is ranked highest among all variables which confirmes the importance of space for information visualization.

A third quality criterion for information visualization can be found in the literature, i.e. *appropriateness*. As Schumann et al. [161] discuss, appropriateness[2] does

[1] Expressiveness and effectiveness actually denote properties of the mapping from data to visual structures. Nevertheless, in this thesis we do not distinguish between visualization expressiveness/effectiveness and property mapping expressiveness/effectiveness, rather we use the corresponding terms interchangeably even though there may be further aspects like interactivity which influence the effectiveness of visualizations only.

[2] *Angemessenheit*, in german

not directly refer to the quality of visualizations but to the computational effort and amount of resources that are necessary to produce visualizations, i.e. the costs of executing the visualization process described in section 2.1.3. While in the context of this thesis special emphasis is put on visualization expressiveness, effectiveness, and consistency, appropriateness is to some extend addressed as well because the visualization pipeline (i.e. the steps that are necessary to transform raw input data into expressive, effective, and consistent software city visualizations) used in this thesis is fully tool supported. The pipeline essentially consists of the SOFTWARE COCKPIT which allows for extracting, processing, and storing software data, and the visualization frontend CROCOCOSMOS which, as discussed in section 6.4, implements all visualization concepts proposed in the remaining chapters, and which provides rich means of interaction to enable a quick exploration of even medium to large sized software systems.

Data as well as visual structures can be either static or dynamic. Consequently, there are four ways of mapping data onto visual structures. As discussed in [39], human perception is very sensitive to visual changes. However, in the context of this thesis, we concentrate on static visualizations, only.

2.1.2 Spatialization

Space is perceptually dominant [39]. It is one of the most effective visual properties for encoding data. When visualizing scientific data that relates to some physical artifact the use of space is obvious: the visualization typically adopts the physical reality. When visualizing non-spatial data, however, the use of space is not so obvious since there is no physical reality that can be referred to. Thus, the question of how to use space for effectively and efficiently encoding data arises. Software systems are inherently non-spatial artifacts, thus information visualization techniques for spatializing data are relevant for software visualization, too.

Skupin and Buttenfield [168] define spatialization as

> ”a projection of elements of a high-dimensional information space into a low-dimensional, potentially experiential, representational space.“

Fabrikant [68] defines:

> ”Depicting information collections as concrete spatial layouts, even when the collections are not themselves explicitly spatial, is an information visualization technique known as *spatialization*.“

Two different types of techniques can be used to spatialize non-spatial data: First, the use of dimensionality reduction techniques allows for mapping high-dimensional relational data onto low-dimensional visualizable geometric space. Second, applying a metaphor, i.e. mapping (not necessarily relational) data to the structures provided by the metaphor.

Dimensionality Reduction Techniques

For visualization only a low dimensional (at most three dimensional) space is available. Mapping high dimensional data onto low dimensional space can be realized by dimensionality reduction techniques like Multidimensional Scaling (MDS) and Force-Directed Placement. The results of both dimensional reduction techniques are placements of entities of a high dimensional space in a lower dimensional space that can be visualized in the form of e.g. two dimensional maps or three dimensional point clouds.

A common goal of these techniques is to find a placement of entities such that similar or somehow related entities are placed close to each other whereas dissimilar or unrelated entities should be spatially separated. This goal is also addressed by the Gestalt principle of proximity (or contiguity) which stresses the need for similar objects to be placed in close spatial proximity as they are perceived as belonging together. We provide examples of software visualization approaches based on dimensionality reduction techniques in the remainder of this chapter.

Visualization Metaphors

Metaphors are a ([93][3]) "pervasive mode of understanding by which we project patterns from one domain of experience in order to structure another domain of a different kind", they allow for ([112]) "understanding and experiencing one kind of thing in terms of another". Metaphors are omnipresent in computer science; desktop, window, port, virus, firewall are commonly used metaphors that associate a particular meaning to the subject they refer to. It is not necessary to know about how a firewall actually works to understand its purpose to serve as protection from external threats.

Metaphors are also used in information visualization where they are called *visualization metaphors*. Averbukh ([8]) defines visualization metaphors as

> "a mapping from concepts and objects of the application domain under modeling to a system of similarities and analogies generating a set of views and a set of techniques for communication with visual objects."

In regard to this definition, the range of visualization metaphors spans from the simplest graphical systems to the adoption of real world physical systems. Chen states that ([45], p.100) "the greatest advantage of using a physical world metaphor is that users can easily understand how the virtual world is structured".

Examples

Dimensionality reduction techniques and visualization metaphors complement each other. An example of this is the *InfoSky* ([82]) visualization in figure 2.1. The InfoSky system adopts the metaphor of a nightly sky to visualize hierarchically structured document collections as stars and constellations ([82]). Based on document

[3] as quoted in [111]

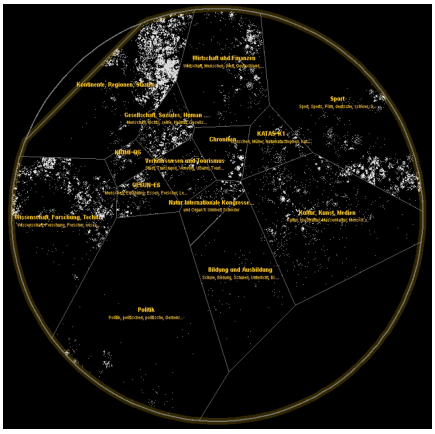


Figure 2.1: InfoSky

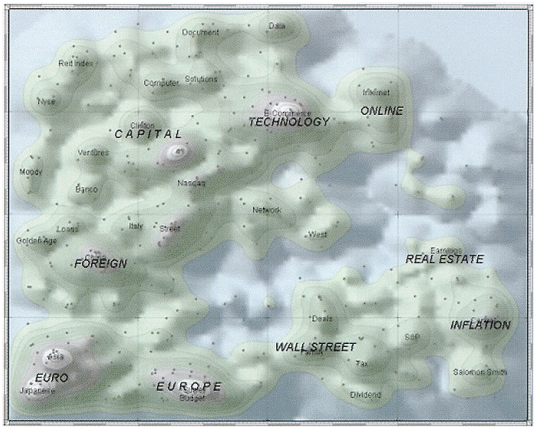


Figure 2.2: Themescapes [170]

similarity stars are put onto a two dimensional map using a force-directed placement algorithm. Polygonal areas surrounding sibling documents are computed using voronoi tesselations.

Landscapes are another metaphor often used for visualizing document collections. An example landscape is shown in figure 2.2: entities are placed on a two dimensional plane in such a way that their positioning reveals information about document similarities. Including elevation to encode entity properties or document density in a particular region yields a thematic landscape-like visualization. Such *Themescapes* were originally introduced by Wise et al. ([199]).

More landscape based approaches have been developed over the years (e.g. [42], [32], [47], [108], [110], [107]). Some of them use reduction techniques to create thematic landscapes of high dimensional data. Others directly map structures in the underlying data onto structures of the virtual landscape. An example of the latter are *Information Pyramids* ([5]) as shown in figure 2.3. Information pyramids show the hierarchical structure of file systems as hierarchically nested plateaus. In contrast to the previous example, the layout of plateaus, i.e. the positioning of recursively contained sub-plateaus and file representations within the plateau, does not encode additional relational data as, for example, in the Themescapes approach.

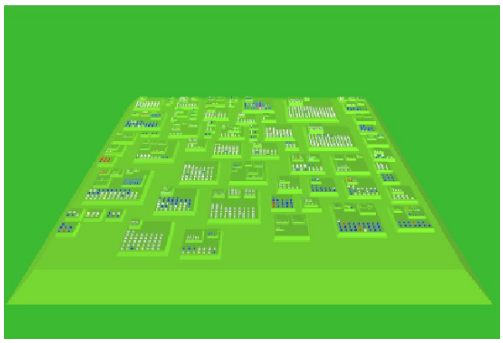


Figure 2.3: Information Pyramid

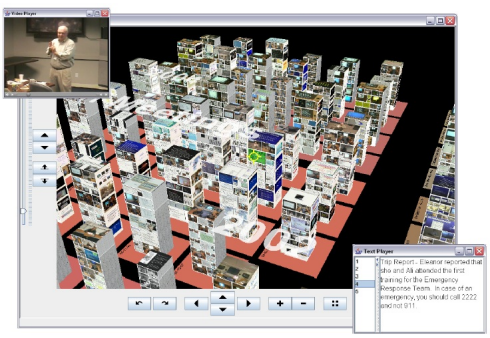
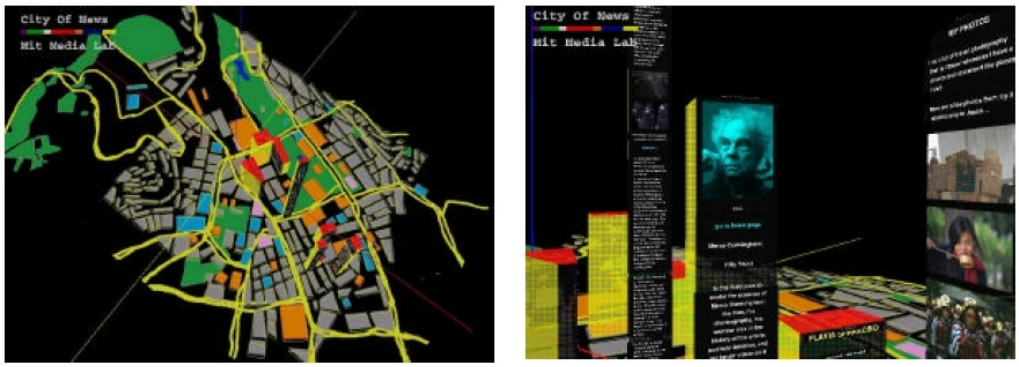


Figure 2.4: MediaMetro [49]

Figure 2.4 gives another example of adopting a metaphor for information visualization. Collections of multimedia documents are visualized as virtual cities. Buildings represent documents. The content of these documents is displayed (in extracts) on the corresponding buildings' facades (which looks similar to windows). Picking such a window opens a corresponding media tool e.g. for viewing videos or images. The layout of the city represents the hierarchical structure of the document collection, it is computed using a rectangle packing algorithm similar to quantum treemaps ([22]).



(a) Aerial Overview (b) Perspective View during Exploration

Figure 2.5: An Internet City in the *City of News* web browser

Sparacino et al. [169] point out a mismatch between the way people communicate about the internet when using spatial concepts like internet address, web site, chat room, and the way they navigate the internet using "the old metaphor of the hypertext and the book". The authors propose using a spatial city metaphor for visualization of and interaction with the web. The *City of News* web browser shown in figure 2.5 visualizes web pages as virtual internet cities. Web pages are represented as buildings with their web page content being mapped onto building facades. Following a link causes a new building that represents the referenced web page to be placed in the internet city. Unfortunately, the authors are not very specific about the details of their placement strategies. Instead, they only state that ([169]) "known cities' layout, architecture, and landmarks are inputs to the program" and that a new building is placed "in the district to which it belongs, conceptually, by the content it carries".

Santos et al. ([1], [158]) adopt the city metaphor in the *CyberNet* project for monitoring NFS related network information. In these cities (figure 2.6) network workstations are represented as districts, workstation disks as buildings, NFS clients mounting the disks as building stories, and file handles of these clients as windows. Anytime a client mounts a disk, a new story is placed on top of the respective building. Thus, building height indicates the number of clients that have a workstation disk mounted so far.

Dieberger and Frank ([56]) give a comprehensive discussion of the design space

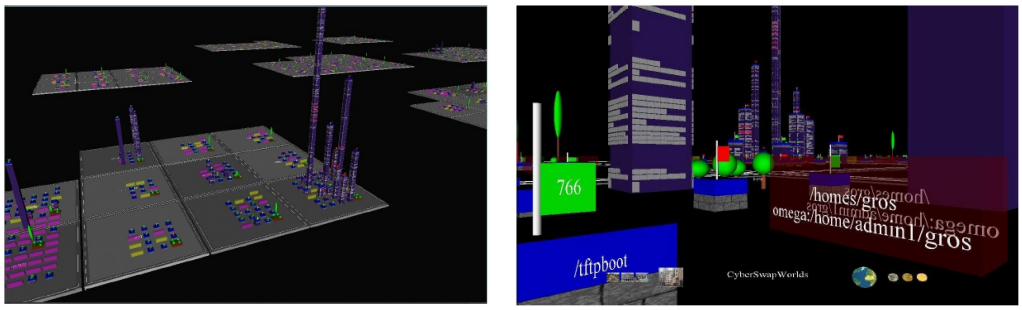


Figure 2.6: The *CyberNet* project

the city metaphor provides. The *Information City* approach they propose is a purely conceptual design meant for exploring and discussing structural and navigational aspects of the city metaphor.

2.1.3 A Reference Model for Visualization

Visualizations transform data of some application domain into visual representations. As discussed in [161], this transformation is typically performed using three sequential processing steps, i.e. a data transformation step that processes raw input data (e.g. by filtering), a mapping step maps processed input data to some visual form, and a final rendering step. Figure 2.7 (taken from [130]) illustrates these processing steps and their respective processing results.

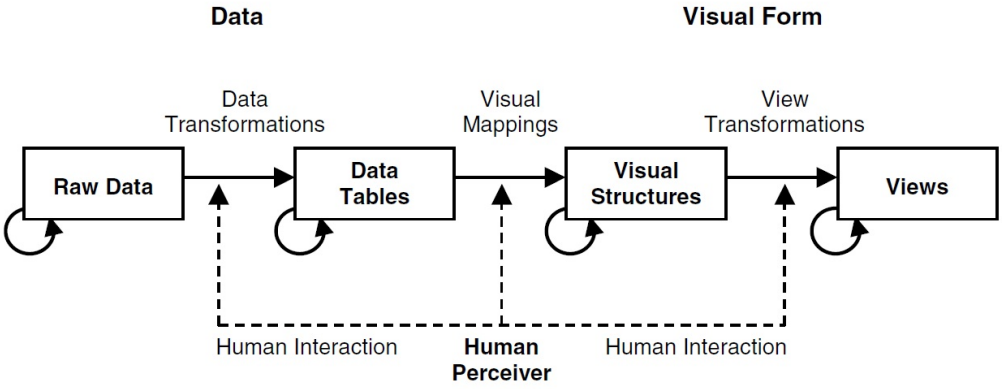


Figure 2.7: A Reference Model for Visualization ([130])

The pipeline starts with raw data like document collections, web pages in HTML format, network data logs and the like. This raw data is processed using e.g. filtering and analysis techniques and stored in some sort of data tables. The content of these are mapped onto visual structures, e.g. by spatialization techniques described above. From these visual structures views are generated using viewpoint controls

like zooming or panning. This latter step is typically performed in an interactive fashion.

Both visualization pipelines (figures 2.7 and 1.1) are compliant with each other although there are slight but important differences: The raw data in figure 2.7 corresponds to the software analysis and product data which are input to our visualization pipeline. Data Tables correspond to the Software Model and the data transformation step is processed automatically by the Software Cockpit. The visual mapping from data tables to visual structures is implemented by two separate steps in our software city pipeline 1.1: First, we *spatialize* a particular subset of the software data to obtain a spatial software model using techniques described in section 2.1.2. Second, we obtain thematic maps by further populating the spatial software model with scenario specific data. This step typically includes the definition and addition of visual entities as well as a scenario specific mapping of analysis data to visual (particular retinal) properties of these entities. Finally, the view transformation step is not explicitly addressed in this thesis as it is not the main focus of the thesis. However, it must of course be implemented in a visualization tool.

The reference model for visualization does not distinguish between a spatial software model and a thematic mapping. For the purpose of this thesis a distinction is necessary specifically because we address inherently dynamic data for which expressive, effective, and most importantly *consistent* visualizations must be designed.

2.2 Software Visualization

This section clarifies often used terms and outlines the field of software visualization. In consideration of the huge amount of software visualization approaches that has been proposed so far a comprehensive discussion of the state of the field is inappropriate in the context of this work. Instead, the following sections discuss taxonomies of the field and point to some up-to-date surveys of software visualizations afterwards.

2.2.1 Terms and Definitions

Knight and Munro define Software Visualization as ([100]) “a discipline that makes use of various forms of imagery to provide insight and understanding and to reduce complexity of the phenomena under consideration”. This definition refers to deriving visualizations from software systems. Regarding the reference model for visualization in figure 2.7, software systems are the raw input data being processed and mapped onto visual structures.

Software Visualization must be distinguished from *Visual Programming* which refers to the process of specifying a program visually using a visual language. Regarding the reference model for visualization in figure 2.7, visual programming relates to manipulating views to define visual structures from which finally raw data, i.e. program code, is derived. A taxonomy of visual programming systems including examples is given by [142].

In earlier literature (e.g. [149]), a distinction is often made between *program visualization* and *algorithm visualization*. Whereas the former refers to the visualization of the actual program source code and data structures, the latter refers to visualizations of higher level algorithmic aspects of software. *Algorithm Animation* denotes dynamic algorithm visualization. Today, these terms are not widely used, anymore. Instead, software visualization subsumes program and algorithm visualization, and beyond that includes visualizations of additional topics as social, economic, or evolutionary aspects as well.

2.2.2 Taxonomies

Taxonomies help in structuring the wide range of visualization approaches and support in selecting suitable visualizations for a given task, or in identifying open fields of research. Several classifications and taxonomies of software visualization have been proposed over the years ([57], [130], [141], [150], [156], [165], [172]).

One of the earliest taxonomies was defined in 1986 by Myers [141] who classifies along two dimensions describing *what* is being visualized (code, data, algorithm), and *how* it is being visualized (static or dynamic). Today, this taxonomy does not cover all topics of software visualization anymore as it actually concentrates on the visualization of programs and program logic whereas recently evolved topics like software evolution are not covered.

During the early and mid 1990s interest in classifying visualization approaches arose and many taxonomies were proposed: Price, Baecker, and Small ([150], 1992) define one of the largest taxonomies in the field consisting of 30 hierarchically arranged classification criteria. Smaller taxonomies were proposed by Stasko and Patterson [172], who each classify along four dimensions aspect, abstractness, animation, automation, as well as Roman and Cox [156] distinguishing between scope, abstraction, specification method, and technique, respectively. Sheiderman ([165]) offers a task by data type taxonomy that is more related to information visualization in general. However, the tasks he defines are known as the information seeking mantra and apply to software visualization as well.

More recent taxonomies are given by Maletic et al. ([130]) and Diehl [57]. Maletic et al. derive a classification scheme containing the dimensions Tasks (why is the visualization needed), Audience (who uses it), Target (What is visualized), Representation (How is it visualized), and Medium (Where is the visualization represented). Diehl [57] suggests a classification that is (similarly to Myers) based on the type of software data to be visualized, but includes the evolution of software systems in addition to their structure and behavior.

2.2.3 Overviews

Visual representations accompany software engineering from its very first days. In [9] Baecker and Blaine outline the early history of software visualization that reaches back to the early uses of diagrammatic representations like flowchart. A

similar historical outline is given by Wettel ([192]) who in addition to achievements from the pre-80s includes more recent advances in software visualization.

The first textbook on software visualization became available just recently ([57]). In addition to providing a large number of example visualizations, the book also covers many visualization topics from perception and cognition to empirical evaluations. Besides this book, there is some more introductory literature to the field, as for example [149], [80], [147].

During the last two decades the field of software visualization received much attention, and the number of proposed approaches increased rapidly. A first collection of such approaches is given in [171] which is in a sense a compilation of selected articles on specific software visualization topics. Recent overviews of particular types of software visualization approaches can be found in [176] and [40]. [176] concentrate on three-dimensional approaches, whereas [40] discuss visualizations of static software aspects. Both collections give an impressive overview of the diversity of today's software visualization approaches.

Many visualization approaches are implemented in publicly available tools. The introductory literature and visualization surveys described above often point out available tools. Additional overviews can be obtained from a few comparative tool studies like [20], [163], [162], and [99] that derive requirements and rank features.

2.3 Spatial Approaches in Software Visualization

Similarly to information visualization, spatializations of software systems often adopt real world metaphors as for examples solar systems ([81]), molecules [131], radar ([52]) and spectrograph ([201]) screens, or even such daily life artifacts like tables and spears [30], or spider nets [148]. There is much research on visualizing software on the basis of such metaphors. In this section, we concentrate on 2D and so-called 2.5D ([16]) (or 2.1D [42]) metaphors of maps, cities, and landscapes, and review the state of the field in spatializing software systems as such.

One of the earliest and most influential approaches for spatializing software systems is the *Seesoft* [66] approach as illustrated in figure 2.8. The essential idea is to provide a visual abstraction of source files by representing source lines as simple graphical lines, thus by abstracting from textual content of source lines. A big advantage of this representation is its code proximity since the resulting file representations resemble low resolution maps of the actual files. Color is used in these representations to depict locations of bug fixes, line oriented statistics like age or number of changes of source lines, or program execution hot spots ([10]).

Several tools and approaches adopt the SeeSoft maps: *Tarantula* visualizes test coverage data [94]. The *Augur* [72] system enhances SeeSoft maps with data on development activity in distributed projects. The *Aspect Browser* [83] allows for highlighting search results of grep-like regular expressions (called aspects). Similarly, the *Aspect Mining Tool* [84] uses a SeeSoft representation to visualize so-called hidden

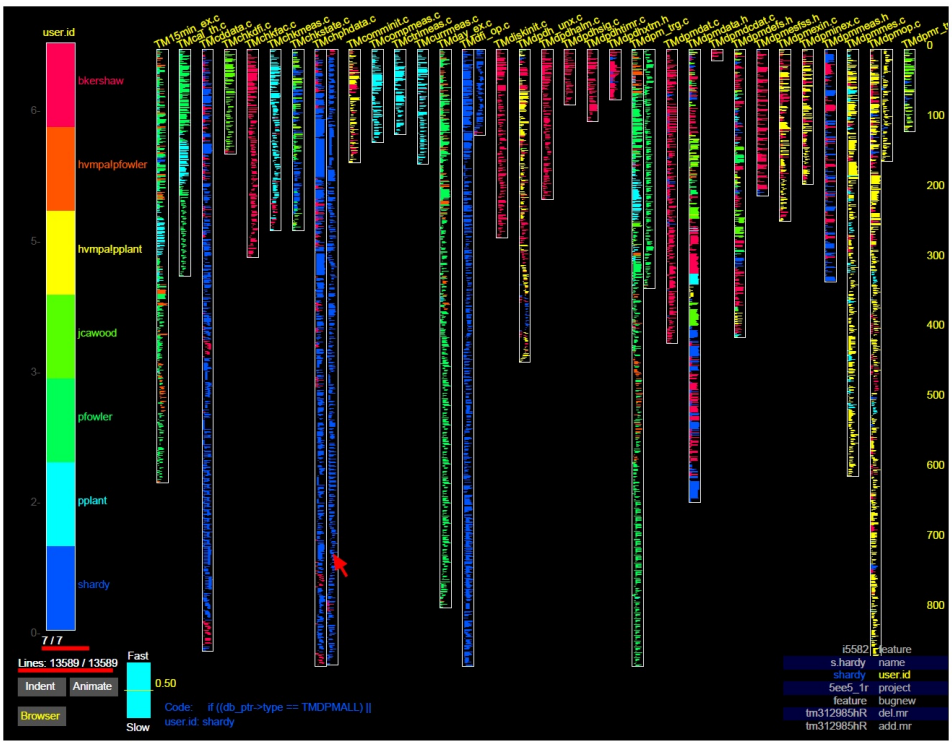


Figure 2.8: SeeSoft based Visualization of Code Ownership ([64])

concerns in the source code. The Bloom system [154] adopts the SeeSoft metaphor for file map representations in both 2D as well as 3D to display file relevant data. DeLine et al. [54] integrate a SeeSoft based representation called *Code Thumbnails* into a development environment to support code navigation. Similar to SeeSoft, Code Thumbnails of several files can be displayed at once forming a so-called *Code Thumbnail Desktop*.

In [55], DeLine et al. describe *Code Maps* that support developers in navigating code. Code Maps as illustrated in figure 2.9 display project documents on several abstraction levels using a semantic zooming technique. Overlays can be added to show search results or execution traces. The layout is computed automatically, but can be adapted by the users. Code Maps are implemented in the *Code Canvas* plug-in for the Microsoft Visual Studio IDE. Code Canvas replaces standard navigational means (tabs and hierarchical overviews) and makes the Code Map the central means of navigation in the IDE.

Support for code navigation also is the goal of *Software Terrain Maps* ([53]) in figure 2.10. In Software Terrain Maps (STM), screen space is partitioned into equally sized tiles. Sets of consecutive tiles depict methods with the number of tiles in a tile set being proportional to the respective method's size. The degree of coupling between each two methods is represented by the distance between the corresponding tile sets, thus computing a STM layout means finding an optimal configuration of tile sets. As the authors discuss, this layout is sensitive to software changes: Removing

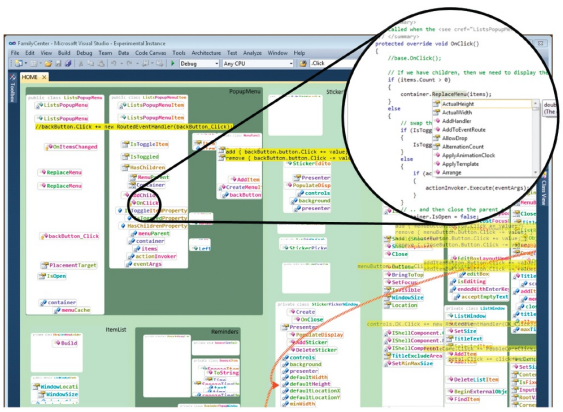


Figure 2.9: Code Map in Code Canvas

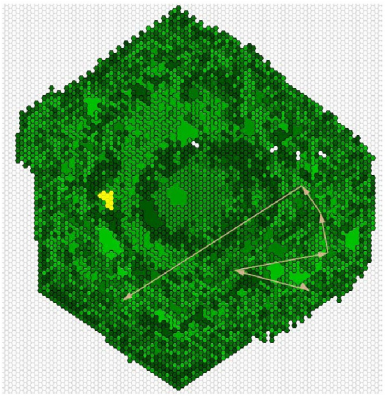


Figure 2.10: Software Terrain Map

methods does not affect the layout as their tiles remain empty. New methods or changes of method sizes, however, may cause large layout adaptations and thus disorientation, and hinder navigation.

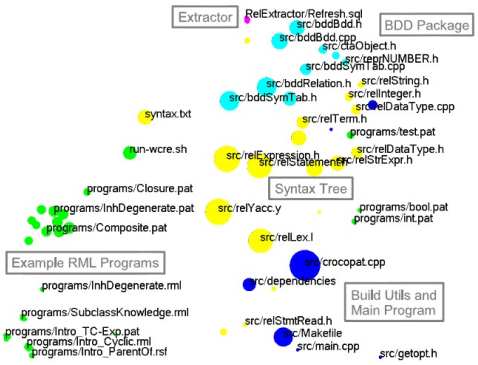


Figure 2.11: A CoChange Map

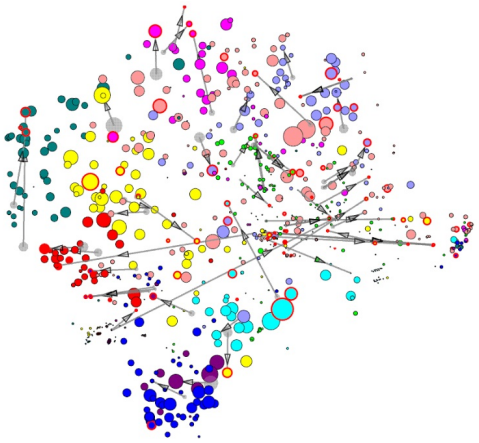


Figure 2.12: Evolution Storyboard

In the majority of cases relational data like couplings between methods form a high dimensional space. Using dimensional reduction techniques this data can be mapped onto a low dimensional and thus visualizable space. Beyer and Noack ([28]), for example, use a force directed placement method to map high dimensional co-change couplings among software classes onto two dimensional maps as displayed in figure 2.11. These maps depict classes that often changed together as cohesive node clusters. Force directed methods allow for using an already computed layout as start configuration of an evolved data set. In [26] and [27] such maps are developed further to so-called *Evolution Storyboards* (figure 2.12). Relational visualizations like these can easily be expanded to three dimensional space as, for example, in [123] and [167] where they are used to motivate refactorings.

Relational layouts are used by Kopsch ([104]) for the visualization of UML RT models. UML RT model elements and their corresponding files are depicted as boxes that are positioned on a 2D plane depending on the dependencies between the model elements. The visualization concepts he proposed, e.g. the use of a colored Voronoi ground plane for the depiction of containment relations, were adopted in the SoftUrbs approach ([125]) for the visualization of software systems as urban structures. In SoftUrbs visualizations, software elements are depicted as boxes which are positioned on a 2D plane such that spatial distance between boxes represents structural dependencies between the corresponding software elements.

The SoftUrbs approach includes additional results from [190] where animation is used when analyzing the evolution of software systems (i.e. when stepping between layouts of successive software versions). Also in the SoftUrbs context, Bartels [18] analyzed if particular characteristics of software architectures like the use of layers can be recognized in the underlying relational layouts. As he illustrated, some architectural characteristics are not always preserved very well in relational layouts. Layouts for layered systems, for example, are typically not laid out sequentially as it is usually done in architecture diagrams. To diminish these differences and to further increase the expressiveness of relational layouts, Bläul [29] proposed the use of additional layout constraints that influence the layout computation in such a way that nodes are positioned within a particular layout region. As a result, the layouts additionally express the overall system structure from an architectural view.

Kuhn et al. ([110], [109]) use Multidimensional Scaling to place software artifacts containing similar vocabulary in close proximity on a 2D map whereas dissimilar artifacts are placed at far distances. The resulting *Thematic Software Maps* (figure 2.13) reveal clusters of thematically similar software artifacts. The authors argue that the project vocabulary is less volatile than software structures, and as a consequence thereof thematic software maps evolve consistently (meaning a low sensitivity to software changes) as the software system is being developed, which allows for an application as consistent platform for locating additional analysis data.

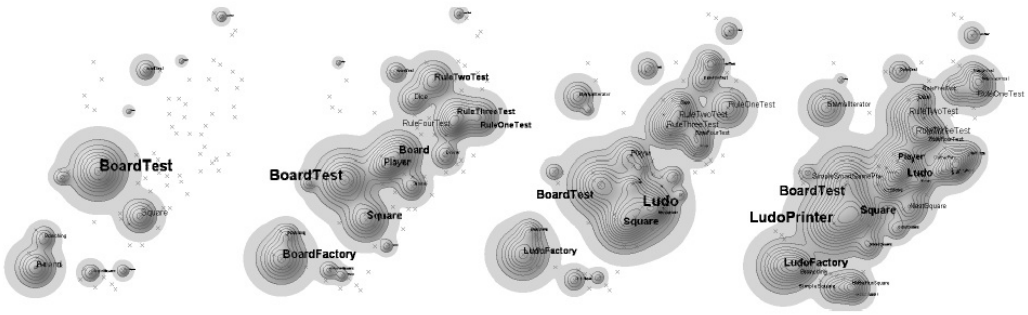


Figure 2.13: Thematic Software Maps for 4 versions of a software system

Based on the SeeSoft pixel representation, the Source Viewer 3D tool (sv3D, [134])

represents source code lines as so-called Poly Cylinders whose visual properties (color, transparency, height, and depth) depict analysis data of the corresponding source line, e.g. execution data [134], nesting depth and control structures [133], or concepts ([203], [202]). Source files themselves are visualized as regular grids of poly cylinders, called containers. An example of a container is displayed in Figure 2.14. Unfortunately, the authors do not describe how containers are laid out, how several containers are arranged as a map, and how these layouts evolve during system evolution.

In [65] Eick et al. describe an approach similar to sv3D called *Cityscape* views. A matrix view relating software modules with software developers is used as 2D platform on which towers depicting the modifications of modules performed by particular authors.

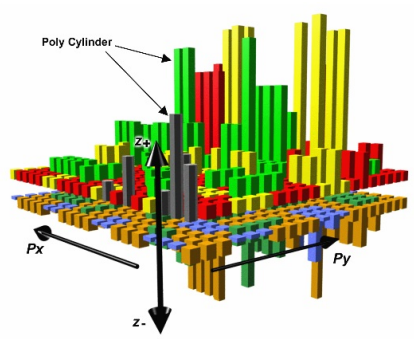


Figure 2.14: Poly Cylinder visualization in *sv3D*

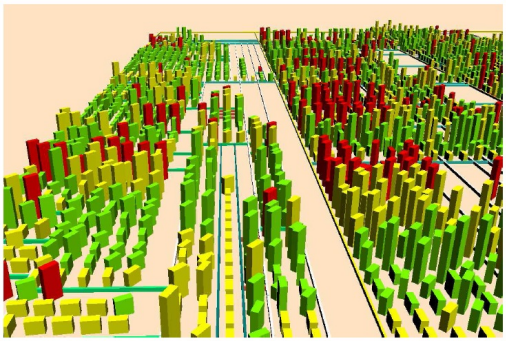


Figure 2.15: Visualization in *VERSO*

Sahraoui et al. [115] describe the visualization framework *VERSO* (figure 2.15) that visualizes classes of software systems as boxes. Visual box properties (color, width, height, and twist) represent class properties. Beyond that, VERSO utilizes space to represent software decomposition using several treemap and sunburst based layout approaches. VERSO also visualizes the software evolution by computing a joint layout for all versions from which layouts for individual versions are derived by hiding removed or not yet existing class. An evolution overview places these layouts sequentially besides each other. In [116] the authors propose layout animation to achieve coherence between successive layouts.

Knight and Munro describe *Software Worlds* ([101], [100]). Software Worlds as illustrated in figure 2.16 consist of countries, cities, districts and fine grained elements like houses, gardens that represent hierarchically structured Java programs with methods, classes, files and directories being mapped onto buildings, districts, cities and countries, respectively. In [43] Charters et al. present a variant of this approach to visualize more abstract software components in *Component Cities*.

Ploix [148] depicts classes and functions as cities and buildings. Functions are categorized as e.g. I/O, numeric, or control functions etc., and grouped accordingly in city districts. Figure 2.17 shows colored districts that contain the corresponding

Figure 2.16: Software Worlds

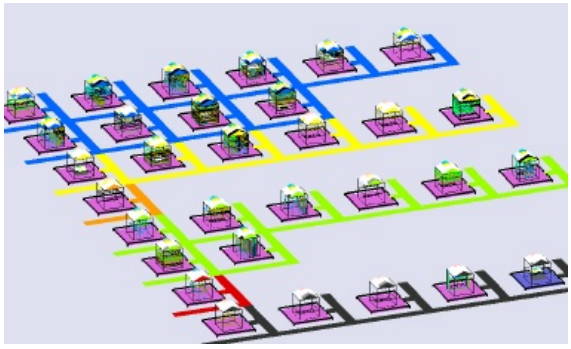
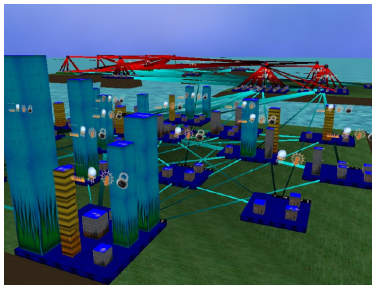


Figure 2.17: Classes as Cities

functions (e.g. I/O, numeric etc.). Buildings consist of roofs depicting numerical analysis data and rooms representing expressions. Inside districts, buildings are laid out in a grid based manner to form an overall quadratic city shape.

In [145] Panas et al. explore the city metaphor on a very detailed level. They indicate the wide range of visual elements (cars, clouds, fire, flashes etc.) the city metaphor provides (see figure 2.18(a)), and they discuss their applicability to visualize static, dynamic, and production cost related information. Elements like trees, streets and street lamps are added to support intuitive interpretation and increase realism. Unfortunately, this approach remained conceptual.

(a) Design Space of Software Cities



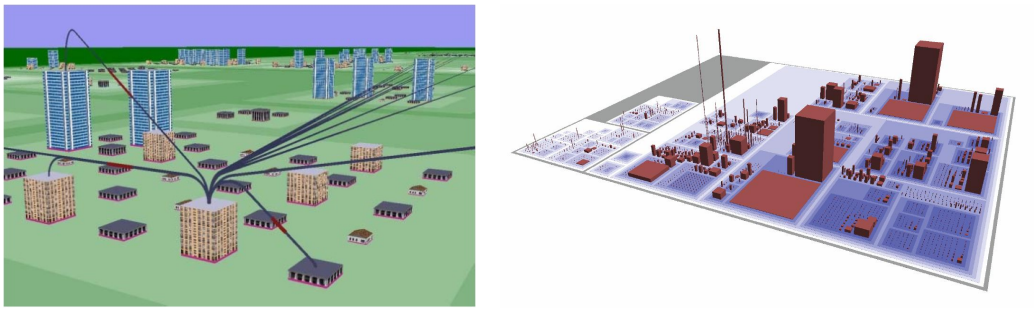
(b) Single View

Figure 2.18: Expressive Software Cities by Panas et al.

Later [146] Panas et al. use the city metaphor to provide single view visualizations designed to address the information needs of different software development and maintenance stakeholders. As illustrated in figure 2.18(b), their cities represent software systems at different levels from coarse-grained directories to fine-grained class member functions. The cities are computed using two methods: Coarse-grained elements are positioned by a force-directed algorithm whereas fine-grained elements (buildings) are laid out compactly in a grid based manner. The resulting cities are augmented with data obtained from analyses (runtime, metric, static/structure, and repository analyses).

The city metaphor has been studied in the EvoSpaces project ([119]) which resulted

in two separate approaches. Dugerdil et al. ([2]) describe an approach in which software artifacts are represented by office buildings or city halls. These elements may be organized in cities whose layouts are computed by several different layout strategies (concentric and chessboard layouts [2], containment layouts [60] using nested areas to depict system decomposition as shown in figure 2.19(a)). Animated solid pipes can interactively be displayed to show element references and their directions ([2]). In [3] this approach is enhanced by a highlighting technique called night view to point out elements involved in a given execution trace. Preceding visualizations of earlier software versions are not taken into account, the approach depicts the state of the system as inputted.



(a) EvoSpaces (b) CodeCity

Figure 2.19: Software Cities in the *EvoSpaces* Project

In the second EvoSpaces approach called *CodeCity* ([194], figure 2.19(b)), software decomposition and artifacts are mapped onto city districts and buildings, respectively. Building size is derived from basic size measures using a boxplot-based mapping technique [193]. Code Cities serve as platform that is enriched with more sophisticated analysis data to support the identification of design problems ([196]), and to gain insight into the structural evolution of software systems ([195]). They present a set of techniques to visualize evolutionary data on both a fine and a coarse grained abstraction level. A time traveling technique allows for stepping backwards and forwards in time and thus allows for visualizing the software system at its different development stages. As shown in figure 2.20, for this purpose each software artifact is uniquely assigned a fixed area in the city. As a result, this approach yields stable layout sequences, which, however, can only be achieved because all versions of the software system are known in advance. As soon as new versions become available, new visualization sequences must be computed, which in turn can differ significantly from the original sequence and the mental models constructed by its users so far.

Similarly to Code City, *Software Maps* by Bohnet and Döllner ([31]) depict the static software decomposition as nested rectangular districts. Software Maps (figure 2.21) are built using a space-filling treemap algorithm which yields very compact visualizations. This high compactness comes at the expense of strongly varying aspect ratios, i.e. ratios between rectangle depth and width that impede perception and increase the risk of misinterpretation. In contrast, Code City is built using a rect-

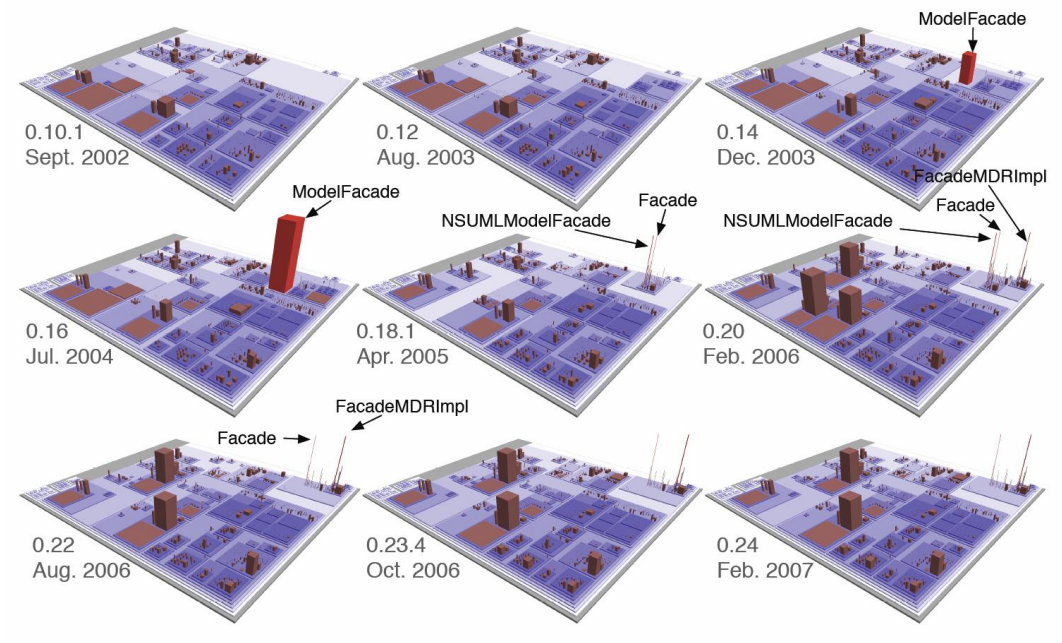


Figure 2.20: Evolution in CodeCity

Figure 2.21: Software Maps

angle packing algorithm that preserves predefined depth and width values. Despite their intended application as monitoring vehicle, the authors fail to discuss layout consistency of Software Maps during software evolution as, for example, Kuhn et al. and Sahraoui et al. do.

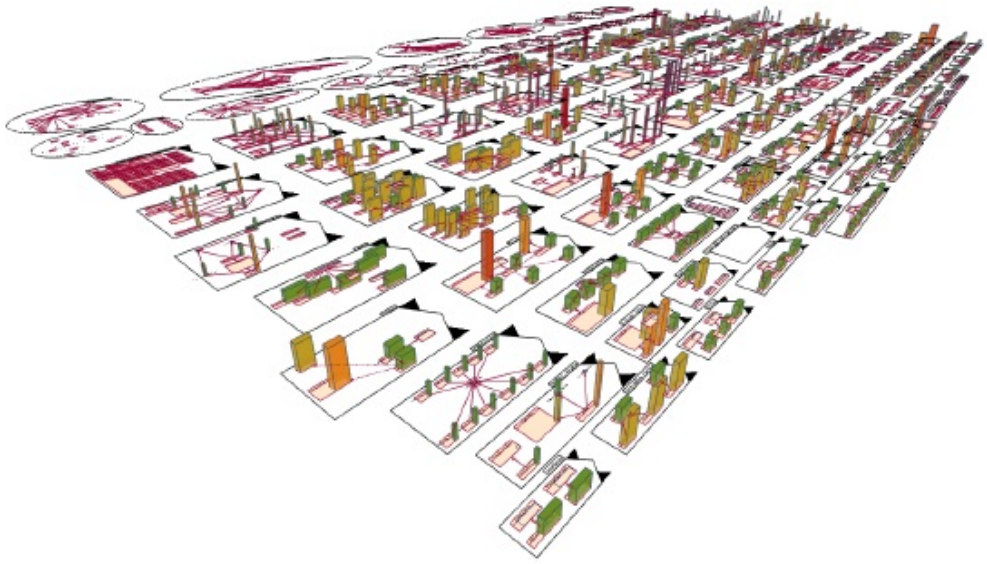
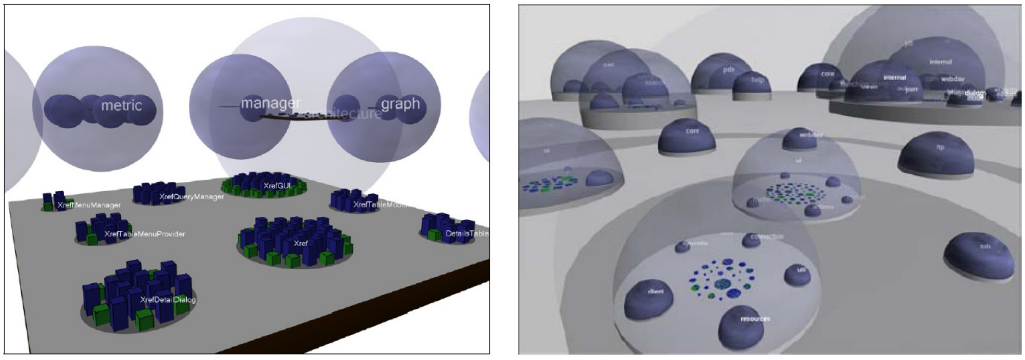


Figure 2.22: UML City

UML Cities by Lange and Chaudron ([113]) depict sets of UML diagrams (like use case diagrams, sequence diagrams, class diagrams, state machines and the like) that are arranged on a two dimensional map[4]. An example UML City is shown in figure 2.22. Properties of model elements (e.g. UML classes in a UML class diagram) can be visualized using height and color of 3D-heightbars that are positioned on the corresponding UML model element. Layout strategies for the overall UML City map are not discussed. Even though the authors address evolutionary aspects in [114] and provide an additional supplementary Evolution View, a discussion of the impact of evolution on UML Cities is missing.

Balzer et al. ([16]) describe *Software Landscapes* as shown in figure 2.23(a). Nested spheres depict the system decomposition, discs and blocks depict classes and methods and attributes, respectively. Since packages may simultaneously contain both other packages as well as classes, spheres contain both other spheres and discs arranged on a 2D plate forming a so-called landscape. The authors point out hierarchical levels of abstraction like continents, states, cities, and finer grained city levels like districts and houses; a hierarchy onto which the hierarchical structure of object oriented software systems can easily be mapped. In [12] a similar approach (shown in figure 2.23(b)) is presented that solely uses nested hemispheres to rep-

[4]A similar visualization concept (not explicitly called UML City) has already been published earlier by the same authors et al. in [175]



(a) Software Landscape (b) Using Level-of-Detail

Figure 2.23: Software Landscapes by Balzer et al.

resent the complete system decomposition from top level packages to the method and attribute level. Both approaches use a proximity based transparency of spheres and hemispheres, respectively, to provide several levels of detail. Both approaches, however, do not address evolutionary aspects.

The term "landscape" refers to a wide range of visualizations from rudimentary 2D graphical representations like [50] to sophisticated landscape-like representations as those by Balzer et al. or Kuhn et al. In a broader sense, the term is also used to denote the set of IT applications of IT companies where they are also called *application landscapes* ([86]). But even these are subject to visualization: In [117] cartographic methods are adopted to depict enterprise application landscapes as so-called *Softwarekarten* (Software Maps, in English).

2.4 Discussion and Summary

Expressiveness and effectiveness are important quality criteria discussed in the literature. Expressiveness denotes the ability of visualizations to express all and only the data that shall be visualized. Effectiveness refers to the quality that allows for efficiently extracting of information. Effective visualizations are easier to understand, require less effort, and cause fewer errors than other visualizations. While expressiveness relates to *what* is visualized, effectiveness relates to *how* it is visualized. In chapter 1, we already discussed a third quality criterion, i.e. the consistency of visualizations during the evolution of the represented data. Consistency refers to the quality of visualizations to support continuous understanding of evolving data. It is an important criterion for the application of software cities during ongoing software development and maintenance phases. Expressiveness, effectiveness, and consistency are the quality criteria addressed in this thesis.

A number of approaches for visualizing software systems as cities, landscapes, or more generally in 2.5D space are described in the literature. These approaches explore the design space of the metaphor and illustrate its high expressiveness. However, only a few of them explicitly address the evolution of the underlying software

systems. In these approaches, evolution is typically observed from a retrospective view which of course allows for creating consistent visualizations very easily. None of the approaches addresses the visualization of evolving software systems during ongoing development, i.e. the visualization of software systems whose development history is not fully known in advance. To better characterize these approaches with respect to expressiveness, effectiveness, and consistency they are analyzed further in chapter 5.

The approaches discussed in this chapter are rather different: Some depict the static decomposition of software systems as nested city districts whereas others depict relational data like similarity of vocabulary or co-change as relational maps or landscapes. Interestingly, there has not been any integration of these approaches into one coherent visualization approach which is surprising particularly because combining these approaches could in fact yield highly expressive software visualizations.

While some approaches explicitly point out their bias towards a particular programming language or paradigm, this is usually no conceptual limitation of the underlying visualization metaphor. Instead, all of the approaches presented above can easily be adapted for other programming languages as well.

Just definitions either prevent or put an end to a dispute.

Nathaniel Emmons

Chapter 3

A Model for Evolving Software Systems

This chapter addresses the first stage of the visualization pipeline described in section 1.3. It discusses the software model that is used for the visualization of evolving software systems as cities. To support a wide range of comprehension and analysis scenarios, the software model has to fulfill some general requirements like language independence, support for different levels of abstraction and more. The specific goal of this thesis is, however, the construction of consistent software cities for evolving software systems to support monitoring and analysis scenarios in the context of ongoing software development and maintenance. This goal has large effects on the design of the software model since it must be able to store data for several software versions simultaneously.

Before defining the software model, two important aspects of software systems are discussed. In the first section the term *software architecture* is clarified, and important architectural topics like architectural patterns and the mapping between software architecture and code are discussed. Section 2 discusses relevant topics concerning software evolution. A general graph based model which covers both aspects is defined in section 3. In section 4 we discuss aspects of populating the model and describe the SOFTWARE COCKPIT, a tool for monitoring evolving software systems.

3.1 Software Architectures

It is common knowledge that during the last decades the size and complexity of software systems increased enormously. The question of how to manage such large scale systems established a software engineering field that received increasingly more at-

tention over the last years, i.e. the field of software architectures. Software Architectures, as Siedersleben [166] discusses, allow for structuring today's millions of lines of code systems. They have impact on the efficiency of the development process, minimize risks, and serve as communication vehicle ([155]), and thus are a key success factor for software development projects ([166]).

Several definitions and descriptions for software architecture can be found in the literature. The SEI[1] maintains a list of community definitions of software architecture that currently contains several hundred descriptions of the term [90]. In general, software architectures are high level, coarse grained structural descriptions of software systems. They structure software systems on much higher levels of abstraction than the software design level or the implemented code. An often quoted and widely accepted definition of software architectures is given in [91]:

> "The fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution."

The definition introduces the notion of *components*. Components are the central structural units of software architectures. They are coarse grained entities that structure software systems on high abstraction levels. For the purpose of this work we assume that components are the only structural entities and thus we use a rather simple architectural meta-model shown in figure 3.1.

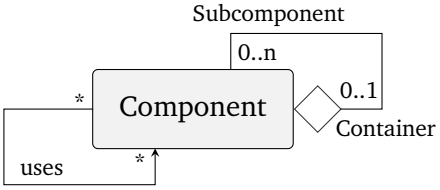


Figure 3.1: Architecture Meta-Model

Components may be composed of other more fine-grained components and thus form a component hierarchy. In this work we assume, that this hierarchy is a tree which means that each component belongs to at most one (more coarse-grained) containing component. Components on the lowest hierarchy level are typically subject to fine grained software design.

As figure 3.1 illustrates, components may also use other components. Such dependencies between components have a direct influence on software quality attributes like maintainability or reusability. For this reason, good software architectures target cohesive but loosely coupled components. Continuously analyzing component dependencies to ensure their compliance with the intended software architecture is an important software engineering activity that allows for an early detection of architecture violations.

[1] Software Engineering Institute at Carnegie Mellon University

3.1.1 Patterns, Styles, and Reference Architectures

Similarly to Object Oriented Design Patters ([85]), architectural patterns describe established conceptual solutions to recurring design or (in this case) architectural problems. In this context the term “architectural style” also is often used in the literature. Following [79], “patterns and styles are essentially the same thing”.

Popular examples of architectural patterns are [76] the Pipes&Filters pattern, or the pattern of Layered Systems. Pipes&Filters systems contain two particular types of components, i.e. *pipes* and *filters*. Filters process input data and output the results to other filters using pipes. Whereas the Pipes&Filters pattern allows for realizing processing chains easily, layered systems are used to structure the system horizontally into *tiers*. Tiers are collections of components. They restrict component dependencies in such a way that components of one tier may use components of the same or lower tiers. More architectural patterns are described in the literature, but as these examples already show, patterns usually make use of own architectural units as (for example) pipes, filters, or tiers.

In contrast to patterns, reference architectures are established architectural solutions to whole classes of applications. Reference architectures define a set of components typically used to realize a particular type of software system as well as the dependencies between these components. A standard example is the reference architecture for compilers which consists of e.g. a Scanner, Parser, Code Generation component. Reference architectures can make use of architectural patterns, too.

Patterns and reference architectures are relevant for this work for two main reasons. First, the software architecture is the main object for several project participants like project managers, software architects, and even customers. Architecture thus cannot be neglected when designing software analysis tools but must be regarded appropriately. Second, for most of these architectural solutions, conventions regarding their visual representation have already been established. Tiers, for example, are usually drawn top-down with relations directing from the top to the bottom. Visual representations of software systems that are built upon such architectural patterns or reference architectures should take the respective placement conventions into account (tiers above each other, pipes and filters alternating from the left to the right, well established reference architectures).

3.1.2 Mapping Architecture to Code

Although software architectures are an important means to structure software systems on coarse abstraction levels, research in the field of software cities has not sufficiently addressed this aspect, so far. This is surprising particularly with regard to the often stated goal of software cities to support the communication of different project stakeholders (e.g [146]). There is only little research that explicitly considers the visualization of software architectures in software cities (e.g. [146], [43]). However, the authors do not clarify their understanding of the terms *software architecture* and *component*, but (essentially) visualize directory structures, only. From a broader perspective, the focus of many software analysis, comprehension, and visu-

alization tools is put on either the architectural level or the code level. Hence, these tools address particular project participants like software architects, developers, or test personnel.

To combine both levels into one visualization a mapping between architectural components and source code units is necessary. Ideally such a mapping is directly encoded in the source code such that each source code unit can directly be assigned to an architectural component. Unfortunately, modern programming languages usually do not provide any constructs to express and verify component definitions, interfaces, access rules and the like ([200]). For example, layered systems organize components horizontally into tiers and restrict dependencies between these tiers from top to bottom. Dependencies from lower to higher tiers are forbidden. Both the organization of components into tiers and the restriction of dependencies between components of different tiers cannot be expressed in today's popular programming languages. While there are some approaches using existing language extension mechanisms like source code annotations to denote e.g. the use of architectural and design patterns ([51]), architecture in the general case is not represented in the source code. Consequently, populating architectural models from the source code relies on additional data e.g. from architecture definitions, e.g. in the form of regular expressions.

3.2 Software Evolution

In this section, aspects of software evolution that are relevant in the context of this work are discussed. We begin with Version Control Systems because they are a fundamental tool for managing evolving codebases. Afterwards we discuss several models used in previous research on software evolution. Finally, we point out a common problem that arises when analyzing software evolution, i.e. the problem of identifying elements in different versions.

3.2.1 Version Control Systems

Version Control Systems (VCS) allow for managing large, evolving code bases in large and possibly even distributed development teams. The core of a VCS is a so-called repository which stores a central development state. Developers obtain their own local working copy from the repository, modify this local working copy, and commit the modifications to the central repository again. Afterwards other developers can update their local working copy to obtain these changes.

Most VCSs support two important mechanisms: Tags and Branches. Tags are an annotation mechanism that allows for labeling specific VCS states. When analyzing the evolution of software systems, these tags are a helpful means as they allow for an easy identification of particular system versions like project milestones or product releases. Branches are a mechanism that allows for parallel system development, for example to implement experimental software functions which shall not be integrated into the system but which on the other hand shall be controlled by the VCS as well. A mechanism that supports such parallel, concurrent development are

branches. The development on a branch does not affect the main system state until the branch is merged with the main development branch In this thesis we consider the main development branch, only.

VCSs are essential software engineering tools since they allow for large-scale collaborative software development. But besides managing codebases by recording and restoring software changes, VCSs increasingly received more attention during the last years in the context of another application scenario: the analysis software evolution ([11]). Much research has been done regarding the analysis of historic software changes, e.g. to analyze growth and change rates ([201]), to understand the structural evolution and erosion [27], to detect non-structural logical couplings between software elements ([75]), determine evolutionary patterns ([118]), understand team structure ([138], [205]) and developer expertise ([140]). Research in this field benefits from the availability of large open source repositories. Today, VCSs are one of the most valuable sources of evolutionary data. In the context of this work, very different data from VCSs (e.g. modification frequencies of software entities) is used to support program comprehension and analysis scenarios.

3.2.2 Models for Software Evolution

Girba [78] distinguishes two coarse directions for modeling software evolution. History-centered approaches address evolutionary properties of the system. The focus here is put on analyzing data that is derived from the evolution of the system and that summarizes and characterizes particular aspects of this evolution. Examples of history-centered approaches are analyses of evolution metrics like modification frequencies or simply the age of software elements. In contrast to history-centered approaches, version-centered approaches focus on the version set. The emphasis is put on comparing versions with each other to answer questions like when particular phenomena (like design flaws) appeared and how they evolved during system evolution. An example of a version centered approach is the evolution matrix discussed in [118]. The evolution matrix depicts software entities and selected entity properties for several versions simultaneously and thus allows for an efficient detection of evolutionary patterns like Pulsar, Supernova, and more.

Girba further proposes the Hismo Meta Model to describe software evolution. Hismo consists of three entities (fig. 3.2): History, Version, and Snapshot. A history collects a set of versions which are ranked to determine an order between them. Snapshot is a placeholder for product metamodels. Although described as a metamodel, Hismo should rather be understood as an extension to these existing product metamodels by the aspect of time. In Hismo, these product metamodels are called *snapshot metamodels* to highlight their static, non-temporal character. For each product metamodel entity, e.g. *Package*, *Class*, or *Method* for JAVA systems, new metamodel entities representing the corresponding history and version are introduced, e.g. *PackageHistory*, *PackageVersion*, *ClassHistory*, *ClassVersion* etc. The approach describes a rather generic history metamodel which allows for adding a temporal dimension to existing product metamodels. Hismo is applied in the Code City approach ([192]) to visualize evolutionary aspects of software systems as cities.

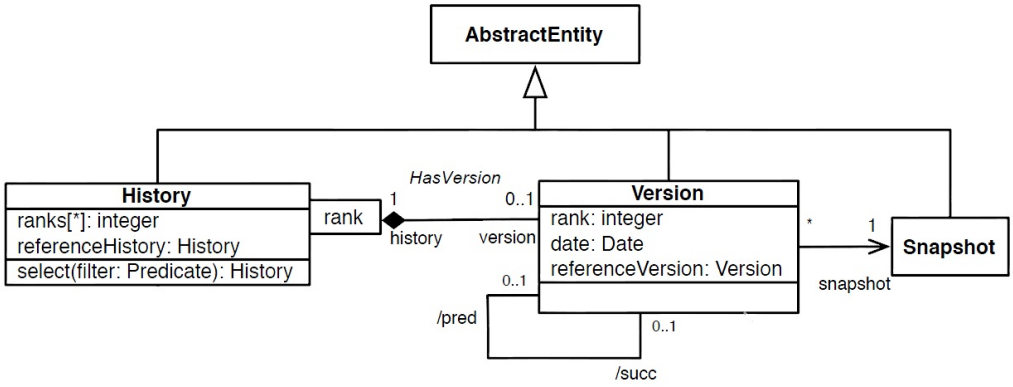


Figure 3.2: Modeling of Evolution in Hismo ([78])

In contrast to the Hismo approach, Hipikat ([185]) is based on a more task-oriented model. Since the tool's goal is to form an "implicitly group memory from the information stored in a project's archives", the model (depicted in figure 3.3) includes very different types of data. Whereas the software system is modeled exclusively on the basis of the files that constitute it, the model includes additional entities like the changes applied to the system, the persons who work on particular changes and more.

Software analysis tools rely on appropriate models that must be designed with regard to the intended application scenarios. Hipikat and similar even larger approaches (e.g. [69]) address very specific analysis scenarios and thus require accordingly specific (and possibly even large and complex) models. The goal of this thesis is to visualize and monitor analysis data from a wide range of different software analysis tools. For this reason, the model to be used here should be generic such that different types of analysis data can easily be integrated.

3.2.3 The Element Identity Problem

The problem of identifying elements in different versions is a general problem of software evolution analysis. The question addressed here is: Given two software elements (e.g. two JAVA classes) in two different versions, how can we decide whether they are identical in the sense that one is a prior version of the other? One simple solution is to recursively define element identity on the basis of their naming, i.e. two elements are by definition identical iff (a) they have the same name and (b) their containing element (e.g. their containing JAVA packages) are identical as well. In this case, however, renaming has of course a large impact on the traceability of elements during software evolution: Whenever an element is renamed its successive version is treated as new element. Unless special care is taken to explicitly model a predecessor-successor relation between both elements, the consequence of renaming would always be a "loss" of historic data.

While renaming is a rather simple system modification that causes problems of iden-

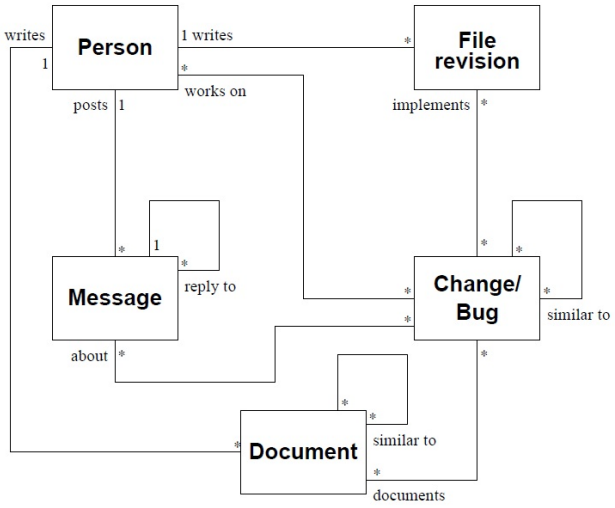


Figure 3.3: Model used in Hipikat ([185])

tifying elements, there are more complex modifications as, for instance, several refactorings like splitting and merging classes as described in [70]. Solutions to this problem are discussed in the literature ([186], [6], [178]). They are based on determining a degree of similarity between elements. In the context of this work, element identity is determined on the basis of element names. We discuss this aspect in further detail in section 3.4.1.

3.3 A Graph Model for Evolving Software Systems

In this section, we define a generic and dynamic model that stores evolving software structures and analysis data. We first define specific requirements for the model, give a definition of the model afterwards, and finally discuss a small example.

3.3.1 Requirements

The model we use in this work must fulfill the following requirements:

- Integration of Software Structures and Analysis Data
  Besides typical software structures like the system decomposition, usage dependencies, or inheritance relationships, the model must be capable of storing analysis data from a wide range of software analysis tools. We restrict ourselves to unstructured analysis data (e.g. software metrics like Lines-Of-Code) rather than complex data like duplicated code (which always refers to sets of software elements).

- Integration of Software Architecture and Software Design
  The model must be capable of integrating both architecture level software elements (like tiers, pipes etc.) and more fine grained design level entities like

packages or classes. Also, it must allow for storing analysis data on both the software architecture level and the software design level.

- Version-Centered Modeling of Evolution
  The model must be powerful enough to store the evolution of software structures as well as the evolution of software analysis data. It must be version-centered to support the retrospective analysis of the evolution of these data (e.g. the analysis of phenomena like design flaws over several system versions).

- Generic Meta-Model
  Today's software systems often are multi-language projects. The model must not be limited to particular programming languages, paradigms, or architectural component types and patterns (e.g. tiers, or pipes). It must allow for an easy integration of many different types of software artifacts, properties, and dependencies.

We now define a dynamic graph model that fulfills these requirements.

3.3.2 Definition of the Graph Model

The core of our software model is a hierarchical graph as described in [143]. This hierarchical graph allows for storing static software structures (like the system decomposition and structural element dependencies) on both the architectural and the design level. The graph model does not yet allow for storing the evolution of these structures, nor does it allow for integrating results from software analysis tools. Therefore, it is extended by an existence and several attribute functions. The existence function allows for describing system evolution. The attribute functions store data obtained from software analysis tools.

Modeling a Snapshot

A hierarchical graph $H = (G, T)$ consists of a directed graph $G = (N_G, E)$ of nodes N_G and edges $E \subseteq N_G \times N_G$, and a rooted tree T. The nodes of T are denoted N_T. The leaves of T are exactly the nodes of G, thus $N_G \subseteq N_T$. The nodes N_G of G represent the elements of the system on a particular granularity level, e.g. classes in object oriented systems. The edge set E represents dependencies between these elements, e.g. usage or inheritance between classes. Higher level dependencies (e.g. between components) are not explicitly modeled because they can easily be computed from edges in E. The rooted tree T represents the system's decomposition into both architectural components (like tiers, pipes, or filters) and design level software elements (like packages and classes in JAVA, or directories and files in C).

Modeling Evolution

The structural evolution of software systems is modeled using a boolean existence function similar to [21]. For each element of the graph (nodes and edges), the existence function stores whether in a particular version the corresponding software

element was part of the software system. The resulting dynamic hierarchical graph H_D is a tuple $H_D = (G, T, R, f_e)$. R is an ordered set of versions. f_e denotes the existence function with $f_e : (N_T \cup E) \times R \rightarrow \{True, False\}$.

While the model is populated, f_e is defined for all nodes $n \in N_G$ (i.e. nodes of the graph G, only) as follows: Whenever a software element is identified in the codebase, the existence function f_e is set $True$ for the corresponding tree node and the current version. If the element is removed from the codebase again (i.e. it is not found in the codebase during the population process) the existence function f_e is set $False$ again for the corresponding graph node and the current version. The node is not removed from the graph.

For tree nodes $n \in (N_T \setminus N_G)$ (i.e. inner nodes that represent software decomposition elements) this is not always possible since in some programming languages these decomposition elements have no direct representation in the codebase (e.g. like packages in JAVA). Instead, they are implicitly defined in source code files. Therefore, based on the graph nodes N_G, f_e is recursively defined for inner tree nodes $n \in (N_T \setminus N_G)$ as follows:

$$f_e(n,t) = \begin{cases} True \Leftrightarrow \exists n' \in succ(n) : f_e(n',t) = True \\ False, else \end{cases}$$

with $succ(n)$ denoting the direct descendants of n. By definition, all tree nodes $n_T \in (N_T \setminus N_G)$ exist iff there is a path in T leading from n_T to a node $n_G \in N_G$ for which f_e returns $True$. Consequently, removing (e.g.) all classes from a JAVA package means that f_e is set $False$ for the corresponding package's tree node as well.

f_e is defined for graph edges as well. Clearly, many types of dependencies require that the corresponding software elements exist (e.g. an inheritance dependency between two classes makes sense only if both classes exist). However, we do not state this as an explicit requirement of the model because in some scenarios dependencies between existing and not existing nodes may be of particular interest, e.g. when analyzing refactorings or restructurings. In such cases, we would use a temporal predecessor/successor relationship.

Modeling Element Properties

There are two types of element properties: First, there are essential properties like the element name and the element type which are always stored in the model because they are necessary for e.g. determining element identity. Second, optional properties which store data obtained from software analyses and which are used in specific application scenarios only. Both property types are stored in the graph by means of node attributes. For each property, an attribute function is defined that returns the value of the corresponding property for each software element.

Since these properties vary over time, the attribute functions additionally take graph versions as input. Thus, for each property, a corresponding attribute function yields

the value of the attribute for a given graph node at a given graph version. Each graph contains a set A of attribute functions $A = \{a_{i \in \mathbb{N}} | a_i : (N_T \cup E) \times R \rightarrow \mathbb{V}\}$. Attribute functions are defined for all nodes and edges of the software model. The target set $\mathbb{V}$ of the attribute functions is not restricted any further, hence attribute functions allow to store a wide range of analysis data.

Properties of software entities are often quantified numerically by software engineering metrics (e.g. [48], [135]). Most of these metrics yield numerical values on an interval or ratio scale. In this work, we do not restrict the scale to be metric but allow for nominal and ordinal scales as well. This allows for supporting a wider range of application scenarios as, for example, the analysis of entity types (e.g. interfaces, implementation classes) or the analysis of authorships. In the latter case, the corresponding property function's target set $\mathbb{V}$ is the set of all subsets of the project author set.

On the basis of the above discussion, we define the model that is used in this thesis, i.e. a dynamic hierarchical attributed graph $DHAG = (G, T, R, f_e, A)$.

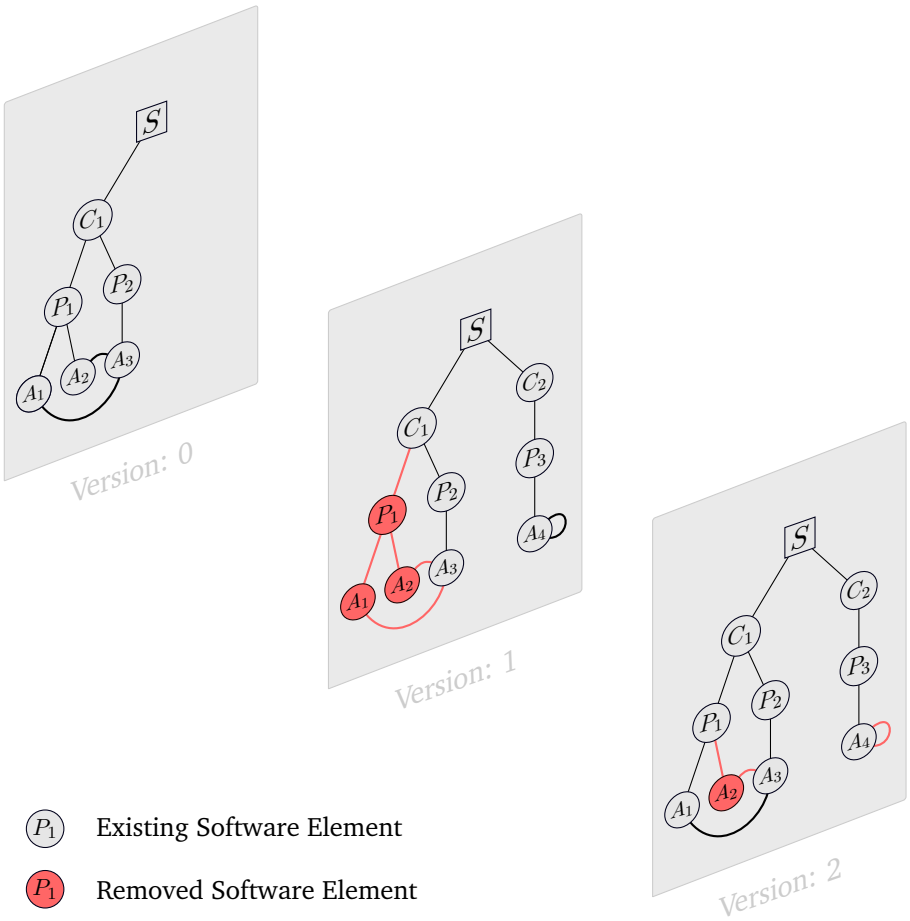
3.3.3 Example

In this section, an example graph for a small fictional software system that evolves over three versions is discussed. Figure 3.4 depicts the structure of the corresponding graph at each version. In its initial version, the system consists of one component C_1 that contains two packages P_1 and P_2. The system evolves over three version such that finally it consists of two components C_1 and C_2 containing three JAVA packages P_1, P_2, P_3, and three JAVA classes A_1, A_3, and A_4. During its evolution, the following changes are applied to the system:

- New Software Elements and Dependencies
  When new software elements or dependencies appear in the software system for the first time, corresponding graph elements are added to the software model. In version 1, the new class A_4 is added to the system. A corresponding node is added to the model. A_4 belongs to a JAVA package P_3 which in turn is assigned to the architectural component C_2. Corresponding nodes are added to the model, too. For each of these nodes, the existence function is set $True$ for the current version and $False$ for all prior versions.

- Removed Software Elements and Dependencies
  In version 1, classes A_1 and A_2 are removed from the system. Once added, nodes and edges remain part of the graph in all following versions. However, for removed elements the existence function is set $False$ for the current version. Since the existence function is set $False$ for all ancestors of P_1, f_e is set $False$ for P_1 as well.

- Reinsertion of Software Elements and Dependencies
  In version 2, class A_1 again appears in the software system. For previously deleted software elements and dependencies the graph already contains corresponding nodes. In such cases, the existence function for those elements is set $True$ again. No new nodes or edges are added to the graph.



The evolution of an example graph over three versions. Nodes and edges are added to the graph when the corresponding software elements appear in the system for the first time. Once added, nodes and edges remain part of the graph even if the corresponding software elements and dependencies are removed from the system.

Figure 3.4: Evolution of an Example Graph as used in the Software Model

3.4 Populating the Model

The model described above is populated from the project source code and from result data of different software analysis tools. Processing these huge amounts of data is a very time-consuming process because the source code as well as software analysis data must be analyzed and integrated for several system versions. Consequently, the software model should not be populated at visualization time. Instead, all data should be processed and integrated into one consistent software model in advance.

A tool that supports such an integration of very different product and analysis data at different development stages into a consistent data set is the SOFTWARE COCKPIT which is described in the next subsection. Afterwards, we briefly discuss the ef-

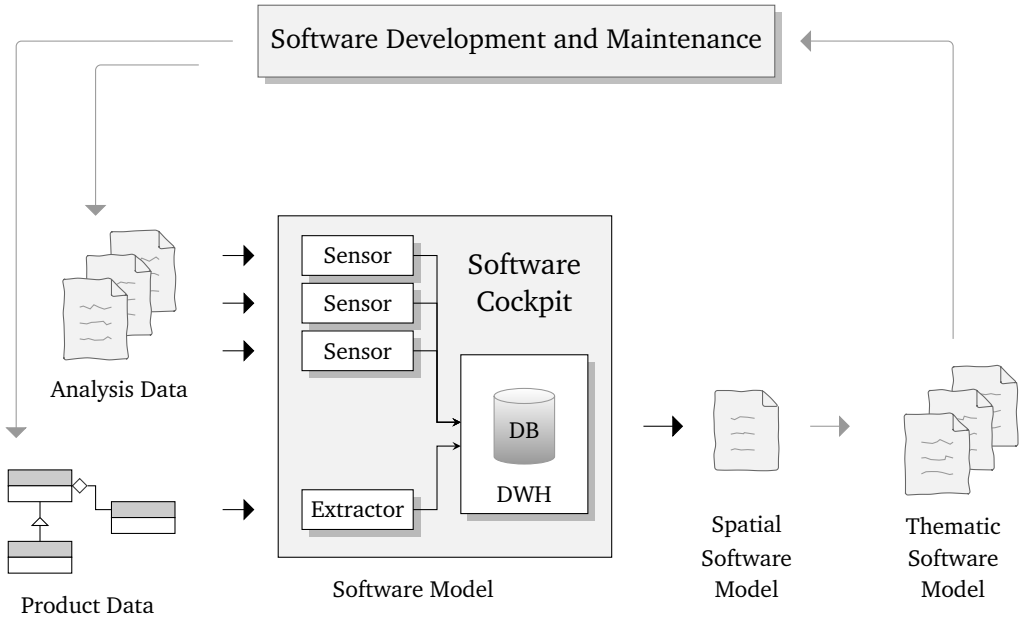


Figure 3.5: Data Extraction by the Software Cockpit

fects of the frequency with which the model is populated, i.e. the model's temporal resolution.

3.4.1 The Software Cockpit

Today's software projects are often heterogeneous projects with respect to e.g. the programming languages that are used for development, or the quality requirements that must be fulfilled. Monitoring such projects to ensure the fulfillment of project schedules and quality criteria requires very different types of analysis tools to be used. The goal of the SOFTWARE COCKPIT (SWC, [24]) is to integrate the huge amount of data from this large set of software tools into one consistent data set. It allows for automatically recording these data during ongoing software development, and supports very different kinds of analyses of the systems' evolution to (e.g.) detect product quality problems or process antipatterns like *Death Sprint* (as discussed in [206]) early during development.

The SOFTWARE COCKPIT has been developed as part of an industrial research project at the Software and Systems Research Group at the Brandenburg University of Technology in cooperation with Capgemini sd&m. Many people contributed to this project, also during master and bachelor theses. Koalick [102] developed a complex data model and a corresponding fact extractor that was capable of extracting and integrating software data from multiple versions of a given codebase into a single data model. This master thesis may be understood as a predecessor of the SOFTWARE COCKPIT. However, the SOFTWARE COCKPIT uses a more comprehensive approach regarding the extraction process, the underlying data model and data storage, analysis features, and representation of analysis results. The corresponding

SWC components are described below.

Software Cockpit Components

As illustrated in figure 3.5, the SWC consists of several components: A Data Warehouse component is responsible for data integration and persistent data storage. Data about the static software structure is extracted by a so-called Extractor component which analyzes the source code with respect to e.g. system decomposition and element dependencies. Additional analysis data is provided by a so-called Sensor component which is a collection of *sensors* that each analyze result data from a particular of software analysis tool. Before we discuss how the SWC is integrated into the development process and how the model is populated, these components and their responsibilities are briefly discussed.

Data Warehouse The Data Warehouse (DWH) component is responsible for continuous data integration and storage. All data that is computed by the Extractor component and the Sensor component is integrated into a consistent data set for several versions of the respective software systems. For this purpose, the DWH persistently stores data into a relational database that is organized on the basis of a three dimensional data scheme. The scheme includes a product dimension to store structural software data, a quality dimension to store the project specific quality model, and a temporal dimension. Additionally, the DWH provides features for further aggregation and evaluation of software analysis data with respect to the project specific quality model (as will be discussed below).

The DWH component was developed by the author of this dissertation. The remaining components Extractor, Sensor Component, and Representation Component were developed by other members of the research group.

Extractor The product model dimension is populated by the Extractor component. The Extractor is responsible for extracting the static structure of the software system from the codebase which is done either via code compilation and AST traversal, or via external analysis tools that analyze already compiled code like the JAVA bytecode. Bytecode analysis is less accurate because some dependencies like references to constant variables are already resolved. On the other hand, compilation quickly becomes a time-consuming process as the codebase grows. Extractors are specific to one programming language each. In case the project is written in several languages, several Extractors are necessary to fully populate the product model dimension of the DWH.

Extractors analyze the project codebase (source code or bytecode). Since the architecture of a software system usually is not directly represented in the codebase (see discussion in section 3.1.2), both the architecture and a mapping between codebase elements and architecture elements (components) must explicitly be defined. In the SWC, it is the responsibility of the DWH to construct the architecture model on the basis of an external architecture description and to populate this model with data from the Extractor component.

Sensor Component The Sensor component allows for integrating results of external software analysis tools into the software model. For each analysis tool, a small program called *sensor* processes the analysis result files and transmits (selected) analysis result data to the DWH component. Sensors typically are responsible for one particular analysis tool, only. For several popular analysis tools like Checkstyle[2], JavaNCSS[3], PMD[4], FindBugs[5], JUnit[6], Emma[7] sensors are already integrated into the SWC. In general, sensors may process any kind of input data rather than only results from analysis tools. For example, a so-called *VCS Sensor* allows for extracting data from version control systems (more precisely from their log files). The VCS Sensor is particularly interesting as it provides history-centered analysis data (as discussed in 3.2.2). For the SOFTWARE COCKPIT, Kock ([103]) developed one particular VCS sensor for the VCS *Subversion* that allows for storing change related data in the SOFTWARE COCKPIT where it can be further analyzed wrt. e.g. change frequencies and the like.

Representation Component The SWC includes a representation component that allows for representing data stored in the DWH using simple tables and diagrams. In his master thesis, Hesselbarth [87] introduced additional scenario specific visualizations that make use of 2D maps to depict dependencies like coupling or co-change between software elements by spatial proximity. As depicted in figure 3.5, the representation component of the SWC is omitted in this dissertation, it is substituted by another visualization tool (i.e. CrocoCosmos) that implements the software city visualizations introduced in the next chapters.

Configuring the Software Cockpit

The SWC must be configured for each project. This configuration step includes the definition of project data like the source code location, the project metamodel (determined by the programming language(s) used), as well as the definition of the intended software architecture, and a quality model.

Architecture Specification For each project, the intended software architecture can be specified in a separate architecture description file. This specification includes the definition of a component hierarchy and a mapping between architectural components and codebase elements. Codebase elements like directories and files implement particular fragments of the system's design. The Extractor component analyzes the codebase and extracts these design elements (like classes, inheritance dependencies etc.). On the basis of the mapping between architecture and codebase, the DWH populates the architecture model by assigning design elements to architecture components as soon as the Extractor component transmits them into the DWH. The mapping rules are based on regular expressions such that codebase elements like

[2] http://checkstyle.sourceforge.net/, last access 23.03.2012
[3] http://www.kclee.de/clemens/java/javancss/, last access 23.03.2012
[4] http://pmd.sourceforge.net/, last access 23.03.2012
[5] http://findbugs.sourceforge.net/, last access 23.03.2012
[6] http://www.junit.org/, last access 23.03.2012
[7] http://emma.sourceforge.net/, last access 23.03.2012

directories and files can easily be assigned to components on the basis of their paths and names.

Quality Model Sensors define metrics on the basis of the data that is provided by their corresponding analysis tools. For instance, an analysis tool may determine violations of code conventions. The result of this analysis typically is a list of code convention violations. The corresponding sensor may define a metric that counts the number of those violations for each software element. A quality model now allows for organizing and combining these sensor metrics. In the simplest case, this organization is just a hierarchical grouping of metrics, e.g. to structure them with regard to priority or analysis type. Beyond that, the quality model allows for combining sensor metrics to derive higher level quality results like bug densities for which bug data and size data from different sensors must be combined.

After all data is stored for a particular system version, the quality model is evaluated by the DWH. The evaluation of a quality model refers to a processing step that includes, for example, the comparison of numerical analysis data with predefined thresholds, the classification of analysis data with respect to different criteria like priority, or a simple filtering of irrelevant data. Additionally, this processing step includes an aggregation of analysis data along the product dimension such that even for coarse grained software elements like architectural components aggregated analysis results can be accessed quickly at visualization time.

Integrating the Software Cockpit into the Development Process

The SWC is a distributed system that allows for managing multiple projects in parallel. All components (DWH, Extractor, and Sensor Component) may run on different hosts. Whereas the DWH runs permanently on a central server the Extractor component and the Sensor component may run on project specific hosts (e.g. on a developer machine or on a dedicated machine for nightly builds) and transmit their results to the central DWH. Thus, each software model that is managed by a DWH can be populated by several distributed components.

Each project must be setup manually. Besides the definition of project specifics like the project metamodel, the codebase locations, the system architecture and quality model, this initial setup also includes the selection of a DWH to be used. When this initialization is done, the model can be populated during so-called population cycles. A population cycle typically includes several steps, i.e. the extraction and storage of software structures, the extraction and storage of software analysis data obtained from analysis tools, and finally an analysis of the quality model as discussed above. Each population cycle refers to one particular software version. Whenever a population cycle begins, first a corresponding software version must be defined in the DWH component which is done by the Extractor component immediately before analyzing the codebase. Afterwards, all data that is transmitted to the DWH is stored with reference to the current software version.

The population cycle can be triggered manually by project personnel or it can au-

tomatically be controlled by a build environment like Ant[8] or Maven[9]. Since these build systems do already trigger compilation, test execution, quality analysis via external tools and many more tasks in a defined order, populating the software model stored in the SOFTWARE COCKPIT is only one additional task to do. If the build system is run automatically e.g. during nightly or weekly builds, the software model can easily be populated automatically as well once the SOFTWARE COCKPIT is appropriately configured.

Depending on the intended application scenarios, different temporal patterns for populating the model are possible, e.g. nightly, weekly, or monthly population, or in case project milestones are reached. We discuss the effects of the frequency with which the model is populated below.

Identifying Elements

If two software elements in two distinct versions have the same path and name, then they are supposed to be identical in the software model (that is stored in the SOFTWARE COCKPIT) they are represented by the same graph node. The SOFTWARE COCKPIT provides no integrated means to identify reengineerings as described above, e.g. using similarity functions like [186]. However, since it is built on a generic graph model, such reengineerings can easily be modeled via edges for e.g. renaming, or split-class if necessary. Their detection is subject to simple heuristics or external tools which can easily be integrated via the sensor system.

Besides identifying the software elements in distinct versions, element identity is a problem in another context of the SWC as well, i.e. during the integration of analysis data provided by the Sensor component. Some analysis tools return result data related to source files in the directory structure. Others may be more precise by providing data for single methods, attributes, or source lines possibly even using proprietary notions to denote these elements. To resolve these inconsistencies, several heuristics for mapping data to software entities are used, e.g. for JAVA systems analysis data that relates to files is typically mapped to a corresponding JAVA class in case both share the same name.

3.4.2 Temporal Resolution

The temporal resolution of the model refers to the frequency with which the model is populated. A high temporal resolution results in fine-grained software models which store software data for many different software versions. In contrast, a low temporal resolution yields more coarse-grained software models which store software data for only few software versions. The span of possible temporal resolutions ranges from single source code edits done by a software developer to long-term project milestones or software releases.

The temporal resolution of the model depends on the application scenarios that are

[8] http://ant.apache.org/, last access 23.03.2012
[9] http://maven.apache.org/, last access 23.03.2012

addressed. While there may be analysis scenarios for which single source code edits are the right temporal resolution, for the application scenarios addressed in this thesis we assume that source code edits are much too fine grained. Instead, several commits (of several source code edits) to a version control system typically mark the most fine-grained temporal resolution we consider.

The temporal resolution has several effects on the software model and thus on the visualizations described in the following chapters: The coarser the temporal resolution, the more time elapses between each two population cycles and consequently the more software changes are applied to the system. Changes to the structure and occurrences of quality problems that happened between each two successive population cycles are not recorded in case they are removed again from the system before the second population cycle. Thus, depending on the temporal resolution, large amounts of changes and analysis data may be invisible if too coarse-grained temporal resolutions are chosen.

A special case concerns history-centered data which is typically stored at a much higher resolution. History centered data like the last date of modification are not restricted to the temporal resolution of the software model, i.e. they do not necessarily refer to the versions stored in the software model. For example, the last date of modification of a software element typically does not refer to the population cycle at or before which the element has been modified last. Instead, it would refer to the revision of a version control system when the corresponding element was modified last. Thus, history-centered data may be based on their own temporal model (VCS revisions or calendar time).

3.5 Summary

In this chapter, the software model that is used in all subsequent chapters for the visualization of software systems as cities is described. The software model captures the static software structure which includes both the system decomposition as well as dependencies between system elements. The software model is based on a hierarchical graph. For JAVA systems, this graph would typically represent the JAVA package hierarchy, classes, and their inheritance, as well as aggregated method-usage, and attribute- and type-access dependencies between classes. For C++ systems, the graph might additionally contain directories, or header and implementation files. In addition to these programming language specific elements, more coarse grained units like hierarchically structured architectural components can be stored as well. This structural software data is enriched by additional data gained from a variety of different software analysis tools (like Findbugs, PMD, and Checkstyle) and process related data (like developer activities, development effort, or test coverage).

The software model is a dynamic model, it is populated from the codebase and results of analysis tools for several software versions. Consequently, the model contains and provides the software systems' structure and product/process related data for a series of different development stages. These development stages can be se-

lected e.g. on a daily, weekly etc. basis, by revision number of a version control system, or whenever a project milestone is reached. In this thesis, when we talk about new or deleted software elements, then new and deleted refers to the previous version stored in the software model. All visualizations described later in this thesis are derived completely from data in the software model.

The model is populated by the SOFTWARE COCKPIT, a distributed tool for recording data (structural and analysis data) about software systems. The SOFTWARE COCKPIT consists of a central Data Warehouse component which persistently stores data obtained from an Extractor component and a Sensor component into a relational database. The Extractor component analyzes the codebase of the software systems to determine their hierarchical software structure and element dependencies whereas the Sensor component processes results from additional analysis tools. By integrating the SOFTWARE COCKPIT into a project's build environment and by populating the model during nightly or weekly builds, a consistent software model can continuously and automatically be recorded during ongoing software development.

All truths are easy to understand once they are discovered; the point is to discover them.

Albert Einstein

Chapter 4

Layouts for Software Cities

This chapter addresses the second step of the visualization pipeline described in section 1.3, i.e. the spatialization of software systems. Spatialization produces spatial software models that extend the software models described in the previous chapter by so-called layouts. Layouts describe the spatial organization of software cities, e.g. the positioning of city elements. They contribute significantly to the overall visualization expressiveness, effectiveness, and consistency. This chapter presents the EVOSTREETS layout approach. The EVOSTREETS approach explicitly takes development history into account. It offers two significant benefits for software cities: First, it provides a higher expressiveness since evolution becomes directly visible in the city structure. This allows for supporting new analysis and comprehension scenarios. Second, it provides a high layout consistency for evolving software systems; EVOSTREETS layouts evolve smoothly during system evolution which allows for using them during ongoing system development and maintenance.

In the first section, we discuss the importance of the layout to the visualization criteria expressiveness, effectiveness, and consistency. Afterwards, we review two classes of layouts which are often used for hierarchically structured data and which are also used for software cities. We then clarify which quality criteria are relevant in the context of this work and discuss how the previously discussed approaches fulfill these criteria in section three. From this discussion we derive the main contribution of this thesis, the EVOSTREETS layout approach, in section four and discuss it in detail.

4.1 Quality Criteria

Important quality criteria for visualizations in general are expressiveness and effectiveness. For software cities, a third criterion is pointed out in chapter one, i.e. the consistency of software cities during software evolution. The expressiveness, effectiveness, and consistency of software cities emerge from a wide range of design decisions and must thoroughly be addressed during their construction including the

selection of suitable metaphor elements and the selection of mappings from data to these elements. In this section, we discuss how these quality criteria can be addressed by the spatial organization of software cities, i.e. by their layouts.

4.1.1 Layout Expressiveness

The layout of software cities is often used to encode fundamental structural software data: Many software cities use nested city districts to encode the system decomposition from coarse-grained to fine-grained software elements. They solely express the system decomposition in the layout and thus all these approaches offer the same degree of layout expressiveness. In contrast, some software city and software landscape approaches express relational data ([110], [125]) like element dependencies. Regarding expressiveness, the different approaches are equivalent insofar as they each represent one particular type of data, i.e. either element dependencies or the hierarchical system decomposition. In this thesis, we say a layout approach is more expressive than another only if it additionally expresses further data that is not represented by the less expressive layout.

Increasing the expressiveness of the software city layout means increasing the expressiveness of the entire software city as well. In this chapter, we first look more closely at typical layouts of software cities and their specific characteristics. The EVOSTREETS approach proposed thereafter provides a higher expressiveness in comparison to often used standard layouts since it additionally represents evolutionary data in the layout. Finally, in chapter 7 we discuss a technique to increase expressiveness, i.e. the integration of several software cities into software landscapes in such a way that coarse-grained component dependencies are represented in the layout.

4.1.2 Layout Effectiveness

Space is among the most effective graphical variables for information visualization, thus it must carefully be regarded when designing software cities. Clearly, the effectiveness of these visualizations is directly influenced by the effectiveness of the underlying city layout, more precisely by the mapping from software data to software city layouts which is implemented via particular layout algorithms. Many different layout algorithms are available. Each of these algorithms has its own characteristics, thus their effectiveness may differ even though they map the same information onto the same visual variable. We will discuss these algorithms and their differences in the next section and further refine the effectiveness criterion afterwards. From this refinement we derive measures to determine the corresponding effectiveness aspects in chapter 5.

4.1.3 Layout Consistency

Visualization consistency is influenced by a number of different aspects, e.g. a consistent use of property mappings. An example of such a property mapping where

consistency must be considered is the visualization of authorships described in chapter 6: Each author is uniquely encoded by a particular color. Obviously, once such an assignment is defined it should be kept throughout all successive visualizations to avoid interpretation errors.

Similar requirements can be stated for spatial characteristics. Holt ([88]) states the *law of position permanence* which requires that corresponding parts of evolving systems should be shown at roughly the same positions and with roughly the same size and shape. Position permanence clearly supports in regaining familiarity with visualizations of evolved systems. However there are additional characteristics like the spatial neighborship of nodes or the spatial order of nodes which must be considered as well. We refer to layout consistency as preserving these characteristics. Two layouts are consistent with each other if they preserve these characteristics. Layout consistency is not a binary layout quality. In chapter 5, we describe several different measures to determine the consistency of different layout approaches.

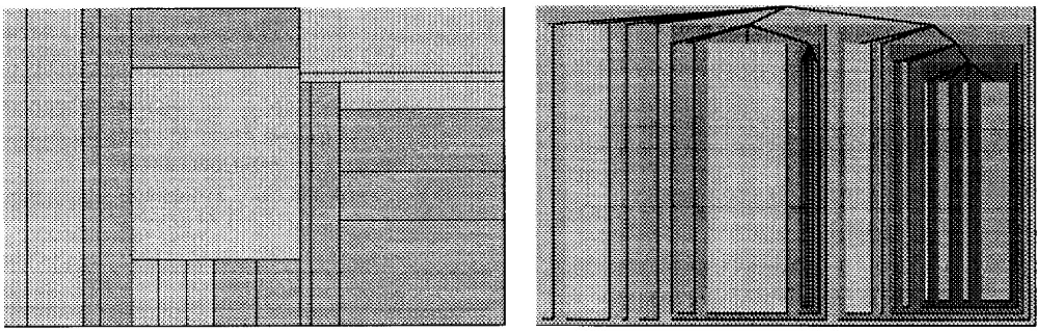
Layout consistency can easily be achieved if the evolution of the system to be analyzed is known which is typically the case if it is analyzed in retrospective. Retrospective visualizations (e.g. [195]) use a supergraph that contains current and past system versions. The supergraph is input to the layout algorithm such that each entity (existing or not) is assigned a unique position in space. Afterwards, particular system states are visualized by simply filtering out not existing structures. The retrospective supergraph approach is a good choice for analysis purposes when, for example, the evolution of particular phenomena in the past must be understood. For ongoing system development this supergraph approach could be used as well. It would yield highly consistent visualization sequences for each system version, but it would not address the consistency between these sequences.

4.2 Related Work

Most software cities or city like visualizations use city elements like nested districts or buildings to represent the hierarchical system decomposition. The logical containment of software elements thus is represented by a spatial containment of the corresponding city elements. Visualizations like these are called implicit tree visualization because they do not use explicit representations (like edges) for depicting containment. Schulz et al. [160] discuss the design space of implicit tree visualizations and provide an overview of current approaches. We discuss two classes of implicit visualizations that are often used for software cities (namely treemaps and rectangle packing approaches) in detail to understand the principles that cause layout properties like consistency or effectiveness to be fulfilled or not. The EVOSTREETS approach described later in this chapter adopts ideas from these approaches.

4.2.1 Treemaps

Treemaps are a popular class of visualizations of hierarchically structured data which is also used for software cities. We discuss the most popular of these approaches in this section. Treemaps depict hierarchically structured data as nested areas. The



(a) Alternating Slice&Dice Treemap (b) Non-Alternating Top-Down Treemap

Figure 4.1: Slice&Dice and Top-Down Treemap

area of each rectangle typically represents a weight of the corresponding node. The essential characteristic of treemaps is their *space-fillingness*: Child node rectangles completely cover their parental rectangle representation. Thus, treemaps make efficient use of the visualization space. All treemaps are space-filling. They differ, however, strongly with respect to the strategies they use for placing nodes. A range of layout algorithms for producing treemaps with different properties has been invented.

Slice&Dice and Top-Down Treemap

The Slice&Dice treemap algorithm was originally introduced for the visualization of hierarchically structured file systems ([92], and [164]). The idea of this treemap algorithm is that the direction along which a rectangular parent node area is split among all child nodes alternates between hierarchy levels: If the root node is laid out horizontally (i.e. its children are placed next to each other from left to the right), then all root node children are laid out vertically. Turo et al. [181] describe a top-down treemap approach which does not make use of alternation.

Both approaches suffer from high aspect ratios. The aspect ratio of a rectangle is the maximum of its width-by-height and height-by-width ratio. High aspect ratios have a negative impact on the readability and comparability of node sizes. Much effort has been spent developing alternatives targeting towards better aspect ratios. In the sequel, the most popular of these approaches are presented.

Strip Treemap

Strip Treemaps, Squarified Treemaps, and Spiral Treemaps are built upon a similar idea: Input rectangles are laid out into a set of *strips*. Each strip is successively populated with input rectangles until the overall aspect ratios of those rectangles already laid out into the strip rectangles gets worse. In this case, the algorithm typically defines a new strip and proceeds by populating this new strip with the remaining input rectangles. All strips together fill the initially defined rectangular

area of the parent node. The major difference between Strip Treemaps, Squarified Treemaps, and Spiral Treemaps is the strategy used for positioning new strips.

Strip Treemaps as described by [23] try to avoid high aspect ratios by laying out rectangles in several strips which all have the same orientation, e.g. horizontally from left to right. An example of a typical strip treemap is given in 4.2 depicting a set of sibling tree nodes. Each of these sibling node rectangles may now serve as bounding rectangle for possibly contained child nodes.

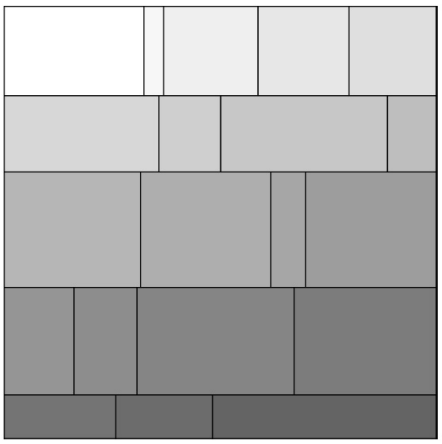


Figure 4.2: Strip Treemap ([23])

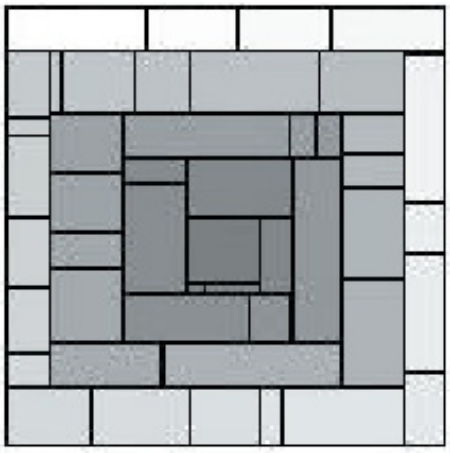


Figure 4.3: Spriral Treemap ([179])

Spiral Treemap

Spiral Treemaps as described in [179] are very similar to Strip Treemaps. The difference is (as the name already suggests) that strips are laid out along the borders of the yet unused area in a spiral manner. Spiral Treemaps were designed to preserve continuity of the input data in the visualization: Nodes which are next to each other in the input data shall be represented by neighboring rectangles in the visualization. An example of a spiral treemap is given in figure 4.3.

Squarified Treemap

Whereas Strip Treemaps arrange strips horizontally below each other, the strategy used for Squarified Treemaps presented by [37] is to layout each strip depending on the aspect ratio of the yet unused space. New strips are placed alongside the smaller dimension of the rectangle, i.e. alongside the left border if the empty area has a larger width than height, or alongside the lower border if the empty area has a larger height than width. In the first case, new strips are populated from bottom to top. An example of this situation is the first strip shown in figure 4.4 containing two rectangles of 6 units size each. In the second case, new strips are populated from left to the right. An example of this situation is the second strip containing rectangles of size 4 and 3, respectively.

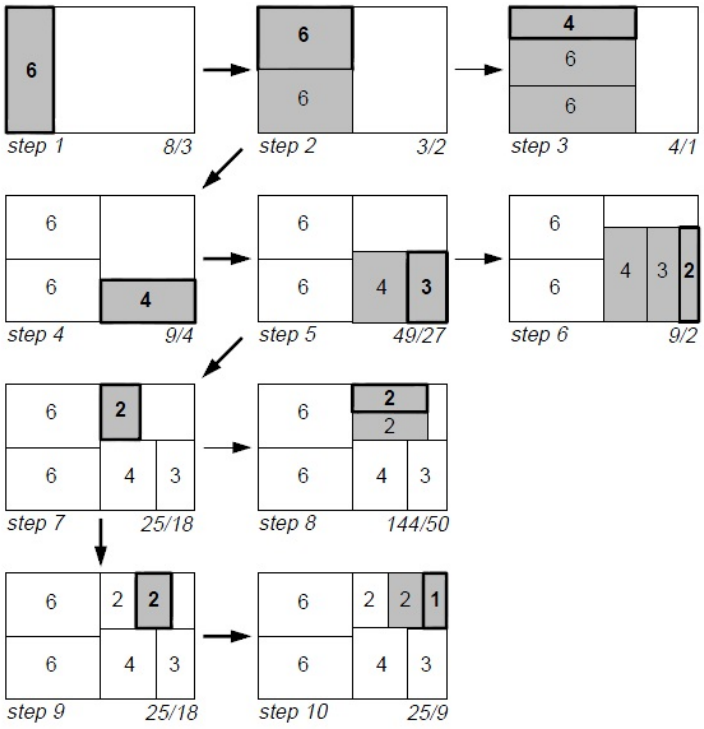


Figure 4.4: Squarified Treemap ([37])

The intended goal of Squarified Treemaps is to produce layouts with better average aspect ratios. For this purpose, the input data is ordered by element size such that large elements are laid out first. If the input data changes this ordering step may cause large discontinuous layout adaptations.

Interesting to note that even the Slice&Dice and the Top-Down Treemap approaches described above can be understood as strip based approaches. The difference to Strip Treemaps, Squarified Treemaps, and Spiral Treemaps is the number of strips actually used: The Slice&Dice and the Top-Down approach use exactly one strip to layout input rectangles whereas the other approaches may use several strips.

Pivot Treemaps

In contrast to strip based treemaps, the main idea of Pivot Treemaps ([23]) is not to layout the input items successively into strips, but to select one specific item R_P called the *pivot* from the input list, to position R_P first, and to arrange the remaining input items relative to R_P. When R_P is placed, the layout space is split into three rectangles R_1, R_2, and R_3 (see figure 4.5). Afterwards, the remaining input items are assigned to those rectangles such that items with smaller input index than R_P are assigned to rectangle R_1 whereas items with larger input index than R_P are assigned to R_2 and R_3, respectively. R_2 is a strip with a variable width. It is successively populated until the aspect ratio of R_P gets worse. Finally, R_1 and R_3 are laid out by

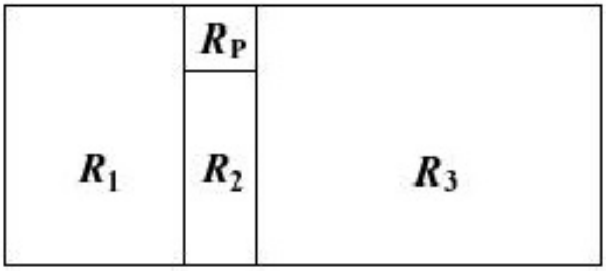


Figure 4.5: Pivot Treemap ([23])

applying the same algorithm recursively.

For the selection of Pivot elements different strategies are proposed ([23]):

- *pivot-by-size*
  The largest input item is chosen as pivot. This strategy is based on the assumption that the largest element is most difficult to place and thus it should be handled first.

- *pivot-by-middle*
  The middle item of the input list is chosen as pivot. Regarding to the authors, this strategy produces more balanced layouts with respect to the number of items in R1 and R2/R3, and is less sensitive to changes.

- *pivot-by-split-size*
  Yet another selection strategy is to split the input list such that R1 and R3 have approximately the same size with respect to the node area. This creates more balanced layouts compared to *pivot-by-middle* if for example the input data is sorted by item size.

Other Treemaps

Beyond these approaches, other treemaps can be found in the literature, as for example Cluster Treemaps [187], Quantum Treemaps [22], Grid Treemaps [159], Cascaded Treemaps ([126], Cushion Treemaps ([184], or Voronoi Treemaps ([13], [15]). The latter differ from all other treemap approaches, as Voronoi treemaps create non-rectangular node representations. Cluster treemaps are supposed to offer good aspect ratios. Unfortunately, the authors fail to provide a sufficiently detailed description of the layout algorithm. Quantum Treemaps refer to treemaps that are constructed by applying a technique called quantization to existing treemaps. The goal of quantization is to produce compact layouts for equally and fixed size input items. Quantized treemaps are used to layout items of uniform size, e.g. image collections. Grid Treemaps define a grid structure which is populated with input items. Similarly to Quantum Treemaps, leaf nodes have a uniform shape and size. Cascaded Treemaps and Cushion Treemaps do not denote particular layout algorithms but specific visualization techniques (*cascading* and *shading*, respectively) for standard treemaps.

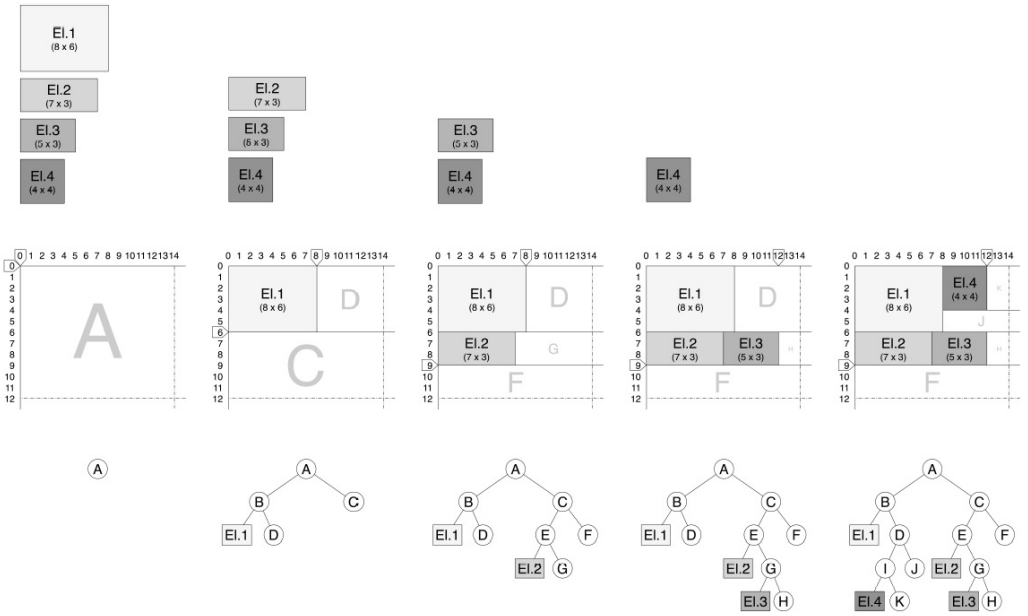


Figure 4.6: Rectangle Packing used in the CodeCity Approach([192])

4.2.2 Rectangle Packing

A major drawback of treemap algorithms is that they adjust the aspect ratio of node representations to achieve compact layouts. Strong variations of aspect ratios have negative impact on the readability and comparability of data items. On the other hand, preserving predefined rectangle dimensions does not allow for placing them in a space-filling manner in the general case. The objective of rectangle packing approaches is to find a layout that satisfies other criteria like aesthetics or compactness. Examples of the former case are the concentric or chessboard layouts used in [2]. In the latter case, the goal of the layout algorithm is to minimize the size of the bounding box that encloses all input rectangles, i.e. it tries to place all input rectangles as close to each other as possible. This problem is also referred to as Minimal Area Rectangle Packing Problem (MARPP, [128]).

MARPP algorithms use different strategies to achieve compactness. While there are many different strategies to achieve compact layouts, we concentrate on a typical MARPP algorithm that is used in the CodeCity approach described in [192]. Figure 4.6 illustrates the algorithm for a set of four input rectangles. The algorithm is based on a binary tree that stores yet unused areas. For positioning an input rectangle the tree is traversed and a best fit container rectangle is determined. If the input rectangle is smaller than the container rectangle the remaining space is split into two rectangles which are stored as unused space in the binary tree so they can serve as container for successive input rectangles. The algorithm requires that the list of input rectangles be sorted by size in descending order ([192]).

4.3 Discussion

4.3.1 Layout Expressiveness

All layout approaches described above represent the same data in the layout, i.e. the system decomposition tree and a particular attribute of leaf nodes that is mapped onto the corresponding rectangles' sizes. Consequently, they all have the same degree of expressiveness. In contrast to non-space-filling visualizations, treemaps have an additional characteristic: In treemaps, the size of each parent node representation exactly matches the sum of sizes of all direct child node representations. The parent representation thus reflects an aggregate value for all child representations. This data is not directly provided in non-space-filling representations like rectangle packing layouts. From this perspective, treemaps may be ranked higher with respect to expressiveness.

A problem with treemaps is that child representations cover their parent representations and that edges in some cases seem to "run into each other" ([13]). We consider this to be an effectiveness issue because it decreases the easiness and error-proneness of extracting encoded data from the visualization. Consequently, expressiveness is not affected by this phenomenon because the data (i.e. the system decomposition tree and leaf node sizes) are still present in the layout.

4.3.2 Layout Effectiveness (Refined)

The effectiveness of software cities strongly depends on the effectiveness of the underlying layout. Effectiveness is a multi-facetted property on both the visualization and the layout level, many different factors may increase or diminish the effort and error-proneness of information reading. To judge the effectiveness of a layout it is thus necessary to consider different aspects. Particularly for treemaps, several such effectiveness criteria and aspects are discussed in the literature, i.e. *(Average) Aspect Ratio* ([23], [37]), *Readability of Hierarchical Data* ([13], [184]), *Order Adjacency* ([188]), *Continuity* ([179]), *Readability* ([23]). While originally introduced for evaluating treemaps, these criteria do apply to MARPP layouts as well, for which another criteria, i.e. the *Compactness*, is highly important. In this section, the above layout approaches are discussed with respect to these layout effectiveness criteria.

Compactness
The most characteristic property of treemaps is their space-fillingness, treemaps are compact visualizations. Compact visualizations offer a high information density and thus increase visualization scalability, support overview scenarios like analysis of the distribution of some phenomenon throughout the system, or allow for comparing distant items in a visualization which is by trend more time consuming and error prone than comparing neighboring items - especially if this comparison requires additional user interaction like panning, or zooming. Compact layouts reduce the average distance between each two items and thus increase the layout effectiveness. Clearly, compactness is not a binary characteristic but a characteristic that allows gradual distinctions.

Readability of Hierarchical Data
Compactness is a reasonable design goal as it has positive effects on visualization effectiveness. Yet high compactness increases the liability to occlusion in both 3D and 2D space. Particularly for treemaps, this phenomenon (increased bias to occlusion) decreases the readability of hierarchical data. Depending on the actual tree structure, containment may not doubtlessly be cognizable in the visualization due to edges that seemingly run into each other ([13]). An extreme example is discussed by [184]: Treemaps turn into regular grids if each parent node has the same number of children and each leaf node has the same size. While this is an extreme example, this effect may occur in every treemap and cause misinterpretation. Thus, even though treemaps provide the same degree of expressiveness (the decomposition data is represented in the layout), they suffer from lower effectiveness caused by this phenomenon.

Although often discussed for treemaps, this phenomenon can occur for MARPP layouts as well. The high compactness of treemaps only increases the problem. The problem can however easily be solved using offsets between hierarchy levels, which in turn decrease the layout compactness.

Variable Aspect Ratios
For treemaps, aspect ratios can be adjusted to achieve space-fillingness whereas rectangle packing algorithms in contrast do not modify the aspect ratio of input rectangles. As discussed in [23], high aspect ratios cause poor visibility and awkward labeling. Some treemaps explicitly address this problem and strive to reduce the average aspect ratio; a value close to one is regarded as optimum. For MARPP algorithms as used in the CodeCity approach, the minimization of aspect ratios is no relevant criterion since these approaches do not modify the aspect ratios of input rectangles at all.

Experiments in [23] and [179] show that the Slice&Dice treemap approach clearly performs worst among all approaches. Whereas the average aspect ratios are typically in the one-digit range, for Slice&Dice treemaps the value is one or two orders of magnitude higher. The best performance is observed for the Squarified treemap approach which in comparison to the Strip treemap approach and Pivot treemap approaches yields values very close to one. Regarding the average aspect ratios, the Slice&Dice approach clearly falls behind.

Besides magnitudes and average aspect ratios, the distribution of aspect ratios is an important but often neglected layout characteristic as well. A wide range of different aspect ratios in the same treemap layout diminishes the comparability of rectangles: Rectangles are easier to compare in size in treemap layouts with constant but high aspect ratios than for layouts with lower but more widely spread aspect ratios. In the former case, rectangles can easily be compared using only the horizontal or vertical dimension whereas the latter requires the consideration of both dimensions for comparison.

Preserving Input Order
As discussed in [23], the input order may have a meaning, for example, items may

be sorted alphabetically. Preserving this order in the layout helps in locating items in the visualization. Therefore, items which are next to each other in the input list should be next to each other in the layout as well. This requirement cannot always be fulfilled since in some approaches the input list is intentionally re-sorted by the layout algorithm to achieve other quality goals. In the Squarified treemap approach, for example, sorting is done based on the assumption that (to achieve better aspect ratios) larger items should be positioned first because they are easier to place. However, the algorithm actually does not rely on this sorting step since it produces valid treemap layouts even if the input list is not sorted by size. Unfortunately, for Squarified treemaps the effects of sorting on aspect ratios were not analyzed by the authors.

Similarly, the CodeCity layout algorithm discussed above does not preserve the input order for the same reason as the Squarified Treemap algorithm. It first re-sorts rectangles by size and afterwards computes a layout for the sorted input list. Though not necessary for layout algorithm, re-sorting again is supposed to yield layouts with higher compactness.

Continuity, Order Adjacency
Continuity (or order adjacency) refers to the property of a layout algorithm to place adjacent input rectangles next to each other in the layout. Discontinuities occur whenever this requirement is not fulfilled, i.e. adjacent input rectangles are not placed next to each other in the layout. Obviously, discontinuities can be caused by re-sorting the input list, but also they can result from the actual layout strategy. For example, a typical discontinuity occurs in Strip treemaps. Strips are laid out horizontally from the left to the right such that the last rectangle of one strip and the first rectangle of the next strip are not placed next to each other even though they are adjacent in the input list.

It is important to note that for all strip based approaches described above (Strip treemap, Spiral treemap, and Squarified treemap) these discontinuities can easily be avoided by using adequate strip placing strategies. In the Squarified treemap approach, for example, strips neither have to start at the lower left corner always, nor do they have to be laid out along particular directions. The discontinuity between each two strips can be avoided by simply starting a new strip where the previous one ended as it is done in the Spiral Treemap approach. These strip placement strategies are somewhat more complex and thus may have negative impacts on layout effectiveness.

[179] investigate the continuity of treemaps and conclude that Slice&Dice treemaps and Spiral treemaps offer best continuity, followed by Strip treemaps and Pivot based treemaps. As discussed in the previous paragraph, the continuity of the Strip treemap approach can easily be lifted to the level that is achieved by Spiral treemaps simply by alternating the layout directions between successive strips. Interestingly, Squarified treemaps clearly fall behind all other approaches even though they are built on the same principles like the other strip based approaches. Its reduced continuity therefore must be caused by the re-sorting step performed on the input data before layout.

From the effectiveness point of view, preserving the input order and continuity are less relevant layout characteristics in the context of this work because the input order usually has no meaning that could be represented in software cities. Both characteristics do, however, impact on the visualizations consistency as discussed below.

Readability
Bederson et al. ([23]) try to determine ”how easily it is likely to be for a person to scan a layout to find a particular item“, a characteristic they call *readability*. A layout is supposed to be readable if it “allows the eye to follow a pattern quickly without having to jump” and if it “allows the eye to jump ahead to the region the user thinks an item will appear”. A high readability would allow to consistently follow a pattern in a predictable fashion when scanning the layout. Obviously, these consistency and predictability qualities are relevant effectiveness characteristics for some application scenarios.

To determine a degree of readability of treemap layouts the authors propose a metric that is based on counting the number of times the user's eye has to change direction by more than 0.1 gradient (ca. 6°) when scanning the layout in input order. Only sibling nodes are considered for this comparison. The metric is used by the authors to compare the readability of different treemap layouts. The results show clear benefits for the Slice&Dice treemap followed by the Strip treemap approach whereas the Squarified and Pivot based approaches clearly fall behind. Similar results were found by [179] who additionally included the Spiral treemap which performed slightly worse than the Strip treemap. Though expected, these results must be handled with care because the authors do not discuss the validity of their metric nor do they provide experimental data supporting their assumption the metric would correlate to a particular kind of readability.

4.3.3 Layout Consistency

Similarly to effectiveness, to determine the consistency of layouts several characteristics must be considered. In this section, we report on results from previous studies characterizing the particular consistency characteristics, i.e. consistency and spatial change. This discussion will provide valuable insights to the mechanics causing inconsistencies between layouts.

Although originally discussed as efficiency criteria, continuity and preservation of input order are important criteria for layout consistency as well. More precisely, discontinuities and re-orderings cause inconsistencies. Discontinuities have strong impacts on layout consistency since they cause nodes to “jump” from their previous to their new position rather than moving smoothly along a continuous pattern. In strip based treemaps, for example, input items are sequentially assigned to strips. If the data evolves, items may be assigned to another (previous or next) strip and thus jump from the end of one strip to the beginning of the next. Depending on the arrangement of the strips and the number of items actually changing their strips between two versions these discontinuities may induce complex layout adaptations

which are difficult to identify and understand.

Preserving the input order increases layout consistency for two reasons: First, re-sorting depends on the size of the input rectangles. Size is typically used to represent a particular property of the corresponding software elements. If this property changes, a differently sorted input list would be processed and laid out differently as well. Thus, changes of element properties can cause inconsistencies if the input order is not preserved. Second, when new rectangles are added to the input list, i.e. if the represented software system grows, the result of re-sorting is a sorted input list where new and old input rectangles are mixed, i.e. the previously sorted input list is now penetrated with new elements. Preserving the input order, on the other hand, allows for attaching new input rectangles to the end of the input list so that old input rectangles can be processed first.

[179] use a simple metric that is essentially based on counting the number of siblings in the tree which are not neighbors in the layout. The results clearly show that the Slice&Dice and the Spiral Treemap perform optimal whereas the Strip Treemap slightly falls behind, followed by the Pivot based approaches which all show similar characteristics. The Squarified Treemap performs worst which is not surprising since it does not preserve the input order, whereas all other treemap approaches do. Since Slice&Dice treemaps use only one strip discontinuities between strips cannot occur. Spiral treemaps use several strips that are arranged in a spiral pattern such that discontinuities cannot occur. Strip treemaps use several horizontally laid out strips between which discontinuities can occur. They can easily be avoided by alternating the layout direction between strips. Squarified Treemaps make use of several strips laid out in a discontinuous manner. Similarly to the Strip Treemaps, the strip arrangement strategy can be adapted such that the algorithm would satisfy the continuity criterion. Pivot based treemaps use a complex arrangement of input items into R_1, R_P, R_2, and R_3. Discontinuities occur between R_1 and R_P, as well as between R_2 and R_3. The algorithm is applied recursively to lay out R_1 and R_3 which causes additional discontinuities.

The results offered by [179] are supported by [23] who investigate changes of treemap layouts for evolving data using a simple metric to quantify changes in rectangle position, width, and depth. The measure is based on the assumption that reading treemap layouts is less time-consuming and less error-prone the less these characteristics of nodes change on average. Similarly to the continuity measure used in [179], the Slice&Dice approach again performs best followed by the Pivot-By-Middle and Strip Treemaps. Squarified treemaps again perform worst. The study includes treemaps only.

The CodeCity packing algorithm described above does not address continuity as a quality property. Rather, rectangles are placed using a best-fit strategy such that a high compactness is achieved. Also, the algorithm does not preserve the input order but requires the rectangles to be sorted by size.

4.3.4 Conclusions and proposed Solution

The goal of this thesis is the construction of expressive, effective, and consistent software cities. These visualization characteristics are strongly influenced by corresponding characteristics of the spatial software city structure, i.e. the layout. Several layout approaches which are in part already used for software cities were described. Regarding their expressiveness, effectiveness, and consistency characteristics we can conclude the following:

- Layout Expressiveness
  All approaches represent system decomposition and a designated leaf node property. Treemaps additionally provide an aggregate of this leaf node property for non-leaf nodes. Besides that, all approaches are considered equivalent with respect to layout expressiveness.

- Layout Effectiveness
  Treemaps in general are compact (space-filling), but suffer from readability of hierarchical data and variable aspect ratios. In contrast, rectangle packings offer a better readability of hierarchical data and avoid variable aspect ratios at the price of reduced compactness. Among all treemaps, large differences regarding continuity and readability can be determined. The Pivot based and the Squarified treemaps clearly fall behind Strip, Spiral, and Slice&Dice treemaps. Slice&Dice performs best with respect to readability, but worst regarding the average aspect ratio.

- Layout Consistency
  Discontinuities and re-orderings cause layout inconsistencies for both treemaps and packings. Thus, the consistency of a layout is not inherently determined by the layout type, rather it has to be addressed by each particular layout algorithm. The Slice&Dice approach and the Spiral approach perform best as they avoid discontinuities and re-ordering. Strip treemaps can be adapted to achieve similar characteristics. Squarified treemaps require re-ordering to achieve better aspect ratios. The minimal area rectangle packing approach requires re-ordering to achieve higher compactness but also places items very discontinuously. Both characteristics suggest a lower layout consistency for minimal area rectangle packing layouts than for treemap layouts.

Layouts for expressive, effective, and consistent software cities make efficient use of space (high compactness), avoid variable aspect ratios, clearly show the system decomposition by avoiding readability problems due to occlusion, avoid discontinuities, and preserve the input order. Considering the discussion above, no approach fulfills these requirements completely, but due to its maximum continuity, preservation of the input order, and the lowest empirical susceptibility to change ([23]), the Slice&Dice approach appears to be a good candidate for software city layouts. Its serious deficiency is the high average aspect ratio which significantly reduces the effectiveness of the layout. However, this problem can be solved using a rather simple idea which is to turn the Slice&Dice treemap into a Slice&Dice rectangle packing.

Figure 4.7 illustrates how this works. The treemap on the left hand side of the figure is turned into a sequence of square rectangles depicted on the right hand size of the figure.

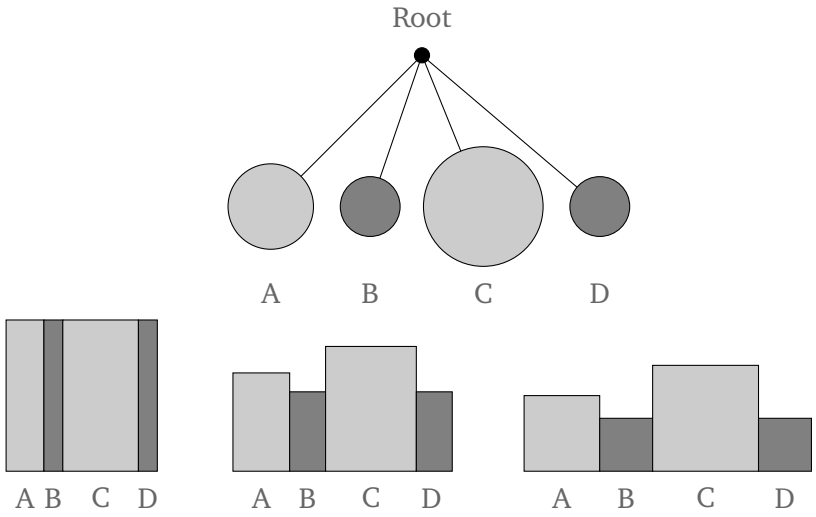


Figure 4.7: Turning a Strip into a Rectangle Packing

Obviously, for each space-filling strip a corresponding sequence of rectangles of arbitrary aspect ratios (squares in figure 4.7) can easily be computed. If in a strip based treemap all strips would be replaced by corresponding rectangle packings than the resulting layout would not be space-filling anymore but it would preserve the underlying treemaps continuity and ordering characteristic. Unfortunately, treemaps are typically constructed top-down by recursively splitting a rectangle into subrectangles. This does not work any longer because the shape of a strip is the result of the packing computation rather than its input. Therefore, these layouts must be computed bottom-up.

For every treemap algorithm described above a corresponding rectangle packing algorithm which is guided by the same layout strategy but which uses predefined aspect ratios (e.g. squares) for all input rectangles can be defined. While effectiveness problems caused by high aspect ratios or a high variance of aspect ratios now can be avoided, another problem arises, i.e. the occurrence of long rectangles (strips). To address this problem, rectangles are laid out using two rows as displayed in figure 4.8. This decision roughly halves the length of each strip and doubles its width. Consequently, the strip becomes more square.

Clearly, this double-sided packing layout can be re-transformed into a corresponding (double-sided) Slice&Dice treemap by simply taking the opposite steps of figure 4.7. This would presumably allow for maintaining the original consistency characteristic but offer improved aspect ratios. However, we are not constructing a ”double-sided Slice&Dice treemap“, rather we are interested in a ”double-sided Slice&Dice packing“ which offers a similar degree of consistency but which on the other hand does

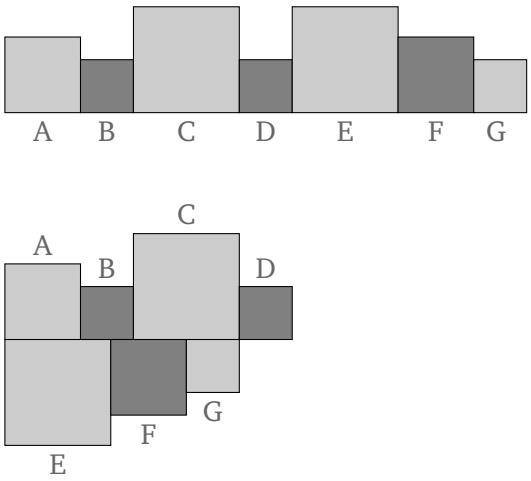


Figure 4.8: Double-Sided Strip Packing

not adapt predefined aspect ratios. We achieve this requirement by simply arranging the double-sided packed strips depicted in figure 4.8 bottom-up in a Slice&Dice fashion (i.e. alternation of layout directions) which is exactly the idea the EVOSTREETS approach described in the next section is built upon.

4.4 The EVOSTREETS Approach

In this section, the EVOSTREETS[1] layout approach is presented. It is influenced by the ideas depicted in figure 4.8 and provides expressive, effective, and consistent layouts. We first outline the basic layout idea and discuss how EVOSTREETS layouts evolve during system evolution. Also we discuss a solution to a particular problem of low layout compactness in this section. The specific construction of EVOSTREETS layouts allows for another new visualization mechanism, i.e. the use of elevation levels and terrains which increases the expressiveness of the layouts. Elevation levels and terrains as well as the visual geographic patterns that result from the use of elevation as a means of expression are discussed in sections 4.4.2 and 4.4.3, respectively.

4.4.1 Basic Layout Approach

The EVOSTREETS approach uses two types of representations: First, inner nodes of the decomposition tree are represented as straight lines. Second, leaf nodes of the decomposition tree are represented by square rectangles. A rectangle is is attached to a straight line iff the represented inner tree node directly contains the repre-

[1] As already discussed in section 2.3, the city metaphor has previously been studied in the EvoSpaces project. Despite their similar names, EvoSpaces and EVOSTREETS are distinct, independently developed approaches. In contrast to the EvoSpaces project, EVOSTREETS denotes a particular layout approach for software cities rather than the overall visualization approach.

Figure 4.9: Mapping Software Structures to Cities in the EVOSTREETS Approach

sented leaf node. The results of this construction are visualizations that resemble a hierarchical street system as shown in figure 4.9.

For JAVA systems, a straight line with attached rectangles typically depicts a JAVA package and its directly contained JAVA classes whereas for C++ systems these layouts might depict the organization of files or classes in a directory structure. The size of the rectangles is typically used to represent any important property of the corresponding software elements. Unless stated otherwise, throughout this thesis the rectangle size always represents the coupling between the corresponding software element and the rest of the system. The coupling of a software element (i.e. the number of incoming and outgoing static dependencies from/to all other classes of the system including method calls, attribute and type accesses, and inheritance relations) plays an important role in many application scenarios since it indicates the potential impact that changes of the element may have to the rest of the system.

System Evolution and Layout Consistency

During a software system's lifecycle the program structure undergoes many changes. New elements are added, others are removed or relocated to other subsystems, element properties like their size or coupling change. As the purpose of the spatial software model is to serve as a stable base representation for the system during its entire lifetime, the overall structure of the spatial software model must not be disrupted by changes of the software model. This is a consistency requirement which is addressed by the following design decisions:

- New tree nodes are represented by corresponding new rectangles or lines. They are attached to the far end of their direct ancestor's line representation. In general, for this purpose the line representation must be extended. Sets of new sibling tree nodes are evenly distributed to both sides of the direct ancestor's line representation.

- If a node is removed from the tree (more precisely, if the existence function is set $False$ for the node at the current system version), then its representation is not removed from the spatial software model. Instead, these nodes remain represented in the layout. To distinguish them from existing nodes, removed nodes are typically represented by another color or a high transparency.

- If the size of a node representation changes (e.g. due to changes of the corresponding property of the software element), all following node representations down the same side of the ancestor's representation are shifted. These shifts can recursively induce further shifts.

- The overall arrangement of node representations is maintained in all subsequent layouts: Rectangles and straight lines remain on the same side of the direct ancestor's line representation that they were initially attached to. The initial spatial order of sibling node representations in the layout is also always preserved.

- The information whether a system element was relocated to another subsystem or removed from the system is not stored in our software models. Thus, elements which moved to another subsystem are treated as removed elements in the context of their source subsystem (the existence function is set to $False$ for the corresponding tree nodes) and as new elements in the context of their target subsystem.

New tree node representations are attached to the far end of their direct ancestor's line representation. Consequently, the overall layout tends to grow at its periphery with older structures concentrating in the center of the layout. Alternatively, new structures could also be attached to the near end of their direct ancestor's line representation such that new structures tend to concentrate at the center of the layout. We discuss this aspect in further detail in section 4.4.2.

Example Figure 4.10 illustrates the effects of these design decisions for our example JAVA system visualized for three different versions. The system grows from initially

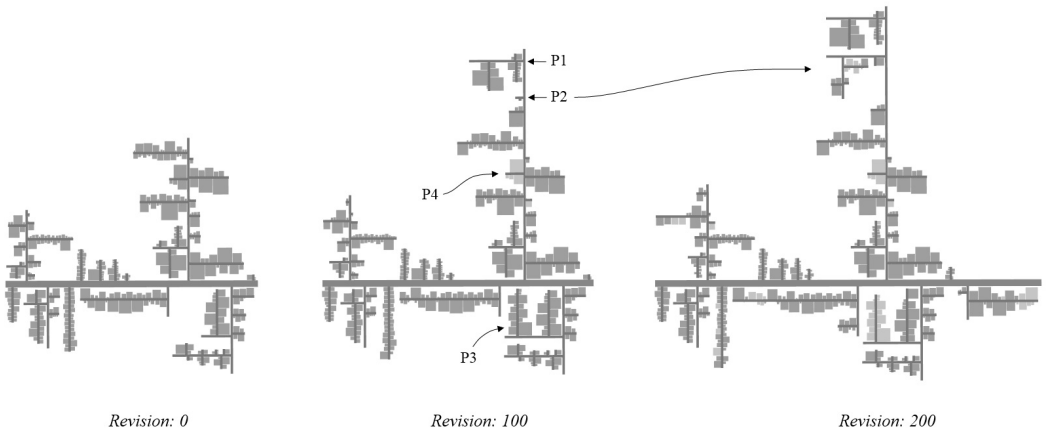


Figure 4.10: Effects on EVOSTREETS Layouts during Evolution

389 classes to 439 classes in revision 100 and to 466 classes in revision 200. The layout represents the system's package structure and classes.

- New elements: Between revision 0 and 100 (among others) two new JAVA packages $P1$ and $P2$ were added to the system. Their representations are attached to the end of the representation of their parent package (a straight vertical line) which has to be extended for this purpose. However, the rest of the layout remains unaffected by this modification.
- New elements: Between revision 0 and 100 another JAVA package $P3$ was added. There is enough empty space available for its representation. Consequently, this modification has no impact on the rest of the layout.
- Growing elements: Between revision 100 and 200 $P2$ grows (a new subpackage is added). Since its representation now requires more layout space, the representation for $P1$ must be shifted upwards.
- Removed elements: Between revision 0 and 100 package $P4$ is removed from the system. This modification has no effect on the overall structure of the layout since representations of removed elements are not removed from the layout. In figure 4.10 these elements are drawn in light gray color. Alternatively, removing the representation for $P4$ from the spatial software model, would cause additional inconsistencies since the representations of $P1$ and $P2$ would (for reasons of compactness) be moved downwards.

As this example illustrates, EVOSTREETS layouts are not fixed in the sense that each representation remains at its absolute position. Instead, changes in the software model may cause local adaptations in the spatial software model that appear as shifts of node representations. However, since the spatial order of neighboring representations is always preserved, the overall structure of the layout is not disrupted from one system version to the next.

(a) Linear (b) Grid (c) Shortest Distance

Figure 4.11: Packing Strategies to Improve Compactness

Increasing Layout Compactness

As discussed in section 4.3.2, layout compactness (i.e. the fraction of the layout area that is actually used for node representations) is an important layout effectiveness criterion. There are two major causes for low compactness in the EVOSTREETS approach. We now discuss one of these reasons, i.e. the occurrence of very long streets. The second reason is discussed later in this chapter in section 4.4.4.

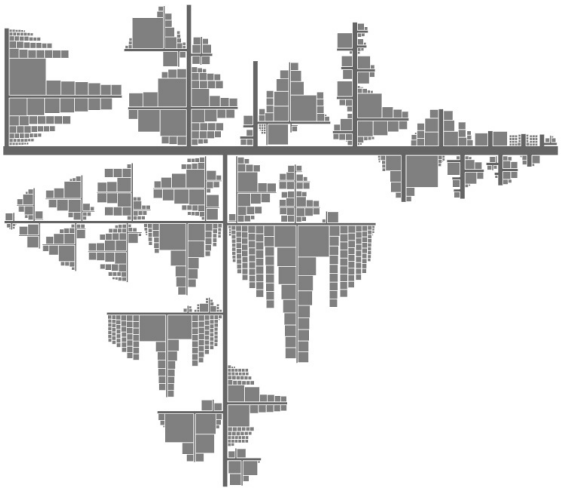
Aligning node representations sequentially along their direct ancestor's line representation as depicted in 4.11(a) in some situations yields very long representation. These long representations depict inner tree nodes which contain a large number of direct child nodes. These nodes cause very spacious layouts. An example of such a layout is given in figure[2] 4.12(a) which depicts the first version of the ArgoUML system (908 classes).

To reduce this particular compactness problem, the EVOSTREETS approach can be extended by rectangle packing algorithms such that neighboring leaf node representations are laid out into rectangular districts. Examples of such packing strategies are depicted in figure 4.11. The linear approach depicted in 4.11(a) is of course a rectangle packing as well. It uses one row on each side of the ancestor's line representation to sequentially place rectangles. The grid based packing strategy depicted in figure 4.11(b) uses several rows on both sides of the ancestor's line representations whereas the Shortest Distance approach illustrated in figure 4.11(c) does not make use of rows at all. Instead, it places each rectangle as close to the origin of the ancestor representation as possible. Clearly, the rectangle packing algorithm used in the CodeCity approach (figure 4.6) can be used here as well.

Depending on the specific packing algorithm used, those rectangle representations that formerly caused very spacious layouts now can be placed more closely. The ArgoUML layout depicted in figure 4.12(a) suffers from a lot of empty, unused space. In contrast, figure 4.12(b) shows the layout of the same system obtained by using the simple grid based rectangle packing strategy (as depicted in figure 4.11(b)). As

[2]The rectangles in both layouts (4.12(a) and 4.12(b)) are sorted by size for reasons of readability, however, this sorting has no effect on this discussion

(a) Linear EvoStreets Layout



(b) Grid based EvoStreets Layout

Figure 4.12: Improving Layout Compactness using a Grid Based Rectangle Packing

this example shows, avoiding extremely long representations can reduce the overall layout size enormously and thus yield more compact layouts. We analyze the impacts of using different rectangle packing algorithms in the EVOSTREETS approach on layout compactness in chapter 5.

Holz ([89]) analyzed the impact of several rectangle packing algorithms on the overall EVOSTREETS layout compactness and stability. However, the algorithms he used strive for high compactness and thus suffer from many discontinuities. In this thesis, we provide a more comprehensive analysis that includes packing algorithms which better balance layout compactness and layout consistency. We analyze the impacts of these algorithms in the EVOSTREETS approach on layout compactness and consistency in chapter 5.

4.4.2 Representing Development History

Determining how a system evolved by visually examining the corresponding layouts is not feasible for real world systems. Instead, evolutionary data is typically depicted by means of color or textures. In these cases, however, the visual parameters cannot be used for representing other (e.g. analysis) data at the same time. In this section we discuss how system evolution can already be represented in EVOSTREETS based spatial software models (i.e. second step of the visualization pipeline) rather than in EVOSTREETS based thematic software models (third step of the visualization pipeline).

As discussed in section 4.3.4, Slice&Dice treemaps avoid discontinuities by laying out sibling input rectangles into linear strips. Additionally, they preserve the input order which allows for laying out new input rectangles without seriously affecting the previous layout by simply attaching them to the end of the input list. The EVOSTREETS approach is based on the Slice&Dice approach and thus shows these characteristics as well. The avoidance of discontinuities and the preservation of the input order allow for a chronological ordering of sibling input rectangles with respect to their creation time. This means that there is a particular direction of growth for each inner node representation (depicted as straight lines) such that in general the farther a child node representation is placed from the origin of its parent node representation the younger the corresponding graph node is.

These specific characteristics allow for a very specific representation of the system evolution: The flat 2D EVOSTREETS layouts can be extended to 2.5D EVOSTREETS layouts by mapping the age of each graph node to the elevation of its representation in the layout in such a way that older nodes are depicted on higher elevation levels than younger nodes. The use of these elevation levels allows for representing evolutionary data (the age of elements) already in the spatial software model, i.e. without restricting the design space of the thematic software models.

Example The EVOSTREETS layout depicted on the left hand side of figure 4.13 represents a small JAVA system which consists of six packages. These six packages were added in two successive versions. In the first version (light gray area), the five upper left packages were added jointly. In the second version (dark gray area), package P

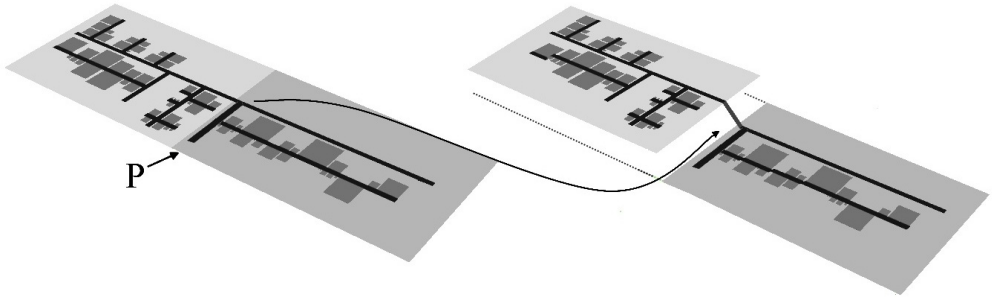


Figure 4.13: Using Elevation Levels to Represent Development History

was added. Mapping the creation time of elements to elevation levels (as discussed above) results in the layout depicted on the right hand side of figure 4.13. The older packages are positioned on higher levels than the younger package P.

Clearly, entire sets of nodes may be added collectively to the same tree node. In such cases, the spatial order of their representations in the layout cannot strictly be interpreted as a chronological order (with respect to the nodes' ages). Instead, these representations form spatial segments for collectively added graph nodes. Within these segments the order has no further chronological meaning. For example, all classes in the only sub-packge of P (figure 4.13) were added collectively. They are positioned on the same elevation level, and their spatial arrangement bears no chronological meaning.

Basically, elevation levels can be used in all treemap and rectangle packing approaches discussed above. However, their effectiveness is strongly influenced by two layout characteristics: First, the continuity of the respective layout directly influences the shape of the overall elevation structure. Continuous layouts offer particular growing directions for each node representation and thus yield rather smooth elevation landscapes. In contrast, discontinuous layouts yield very bumpy elevation structures which complicate layout interpretation. Second, high layout compactness (as offered by treemaps) causes much occlusion which reduces the visibility of elevation levels in the layouts.

Creating a Terrain

As illustrated in figure 4.15(a), understanding the overall elevation structure of EVOSTREETS layouts and thus understanding the evolution of the represented system is difficult, because elevation it is not directly readable in the visualization. Instead, it must be inferred from the overall structure of the layout, especially by analyzing the elevation steps. The idea to solve this problem is to add a terrain to the city that carries these structures and that makes elevation visible very much like mountains. The result of adding such a terrain to an elevated city is shown in figure

4.15(b).

The terrains are constructed very similarly to an approach described in [14] for the construction of implicit surfaces. Implicit surfaces are described by a set of arbitrary generator objects like circles, lines, or triangles, which influence a global density field. As illustrated in figure 4.14, an implicit surface is defined as the set of those points where the density field equals a certain predefined threshold τ. In this thesis, leafs and inner tree nodes representations (which both are depicted as rectangular areas) are interpreted as generator objects which influence the global density field depending on their position and height. In contrast to [14], the resulting global density field is not compared against a threshold to compute an implicit surface. Instead, density is interpreted as height which yields smooth terrain surfaces.

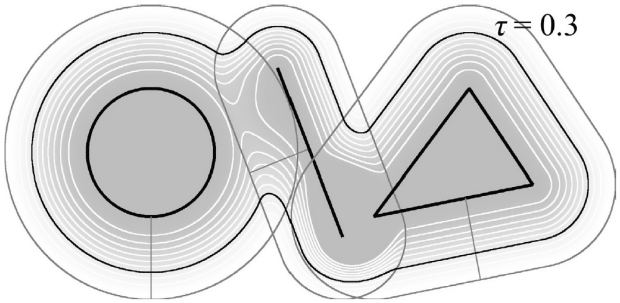


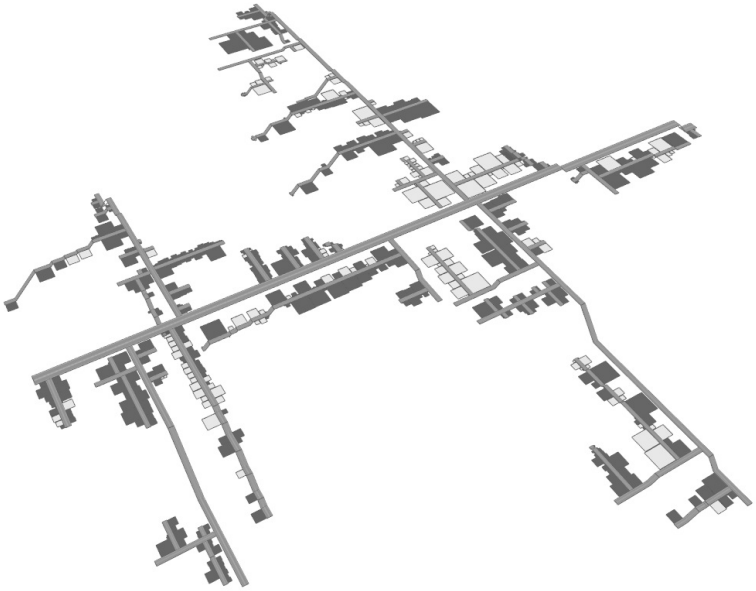
Figure 4.14: Implicit Surfaces by [14]

The adoption and implementation of this concept has been part of a master thesis performed at the research group. A detailled description of the algorithm can be found in [174].

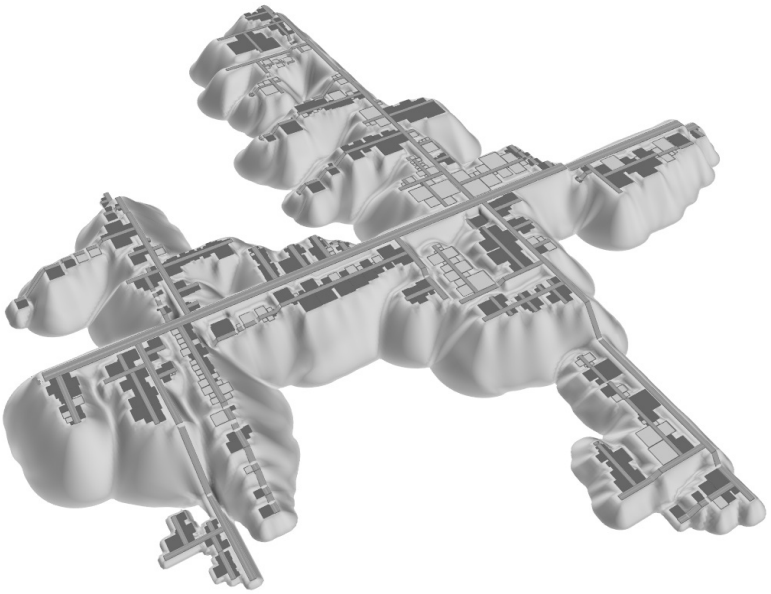
Growing Direction

The EVOSTREETS approach is based on three decisions. First, new structures are added towards the far end of their parental street representation. Consequently, the city tends to grow at its periphery. Alternatively, new structures could be placed at the near end (the beginning) of their parental representations. Then, however, the city would tend to grow from its center which, as discussed above, would cause more instabilities to older substructures of the city because these would be shifted towards the outside independent of whether they have been modified or not. In this work, we do not consider this option any further.

The second decision concerns the mapping of age to elevation levels: As already discussed by Tauer ([174]), new structures can be placed above older structures, or they can be placed below older structures. In the first case, new structures are placed on higher elevation levels than older ones. In the latter case, old structures are placed on higher levels than newer ones. As a result, the cities grow upwards or downwards, respectively. Finally, the third decision concerns the elevation of the landscape's ground level, i.e. whether new structures or old structures shall



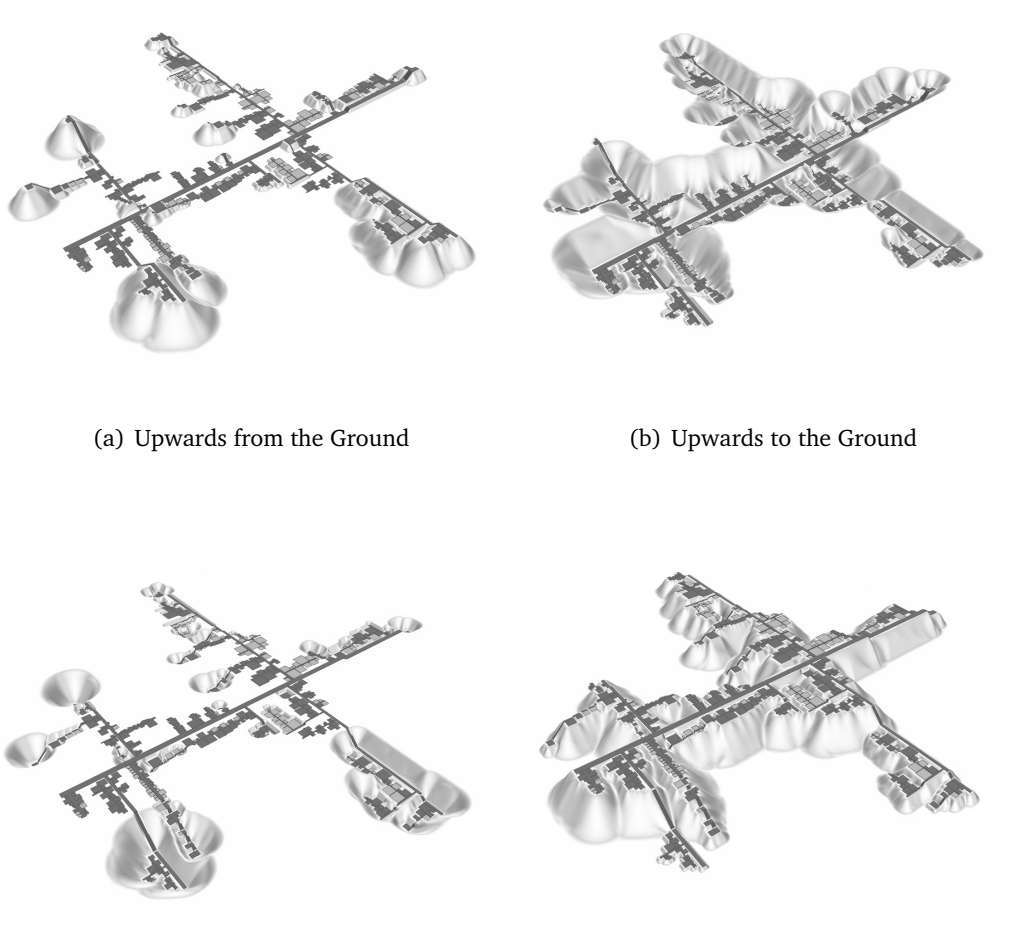
(a) Perspective view with elevation



(b) Perspective View with terrain

Figure 4.15: Using terrains to depict Evolution

be placed at height zero. The latter two decisions results in four layout options illustrated in figure 4.16.



(a) Upwards from the Ground

(b) Upwards to the Ground

(c) Downwards from the Ground

(d) Downwards to the Ground

Figure 4.16: Growing Directions of the Terrain

As figure 4.16 illustrates, the layout options result in rather different software cities. Each of them emphasizes a particular aspect of the underlying software system's evolution:

- Upwards from the Ground
  The city grows upwards with the oldest structures remaining on the ground plane. As the system evolves the city is successively surrounded by hills onto which new structures are placed. This option emphasizes new structures as they are placed on high mountains surrounding the city whereas older structures successively become occluded the more new structures are added. The older the whole system becomes, the larger the mountains for new structures

will be. Also, new structures may cause much visual change when they are first represented in the city and thus cause disorientation.

- Upwards to the Ground
  The city grows upwards with new structures being placed on the ground plane. Consequently, older structures must successively be placed on lower elevation levels below the ground plane, i.e. they sink into deep valleys and thus successively become occluded. From this point of view, the city actually does not grow upwards but rather sinks into successively deepening canyons.

- Downwards from the Ground
  Growing downwards means that older structures are placed on higher elevation levels whereas newer structures are placed on lower elevation levels. In this option, if old structures are placed at the bottom elevation level then new structures successively dig themselves deeper into the ground, they disappear in deep valleys and become strongly occluded whereas the older structures remain visible.

- Downwards to the Ground
  If the city grows downwards with the newest structures being placed on the ground elevation level then the older structures successively become elevated. This option creates a coherent massive mountain shape that emphasizes older structures as they are always visible whereas newer structures surrounding the mountain panorama on the ground may be occluded by the mountain shape.

Growing upwards causes the city center to be occluded by newer structures, either by disappearing in deep valleys or by getting surrounded by high mountains. This may be helpful if particular focus is put on new software structures, e.g. when the development progress is to be monitored or when properties of new structures shall be monitored. On the other hand, when growing downwards older structures remain visible which supports for monitoring or keeping track of the quality of existing codebases.

In the EVOSTREETS approach, we generally use the combination of Growth-at-the-Periphery and Downwards-To-The-Ground since this combination creates consistent cities with a coherent shape. A similarly coherent shape, however, could be obtained if new structures were placed at the beginning of their parental representation on higher elevation levels, i.e. for the combination Growth-at-the-Center and Upwards-From-The-Ground. Similarly to volcanoes, the resulting cities would grow higher at their centers and shift older structures outside.

4.4.3 Visual Patterns

The EVOSTREETS approach basically is a strip based rectangle packing approach. It uses a continuous layout of elements along one dimension and additionally represents them in chronological order. It thus allows for combining all elevation levels into a landscape with a rather continuous terrain on which the city elements, streets and buildings, can be positioned. For layouts based on discontinuous rectangle packings no such unique direction for growth is available, thus using elevations would

yield very bumpy landscapes. Regarding treemaps, for all strip based treemaps such a growing direction exists as well. But due to their compactness and the resulting occlusion elevation would not be readable in these visualizations in general.

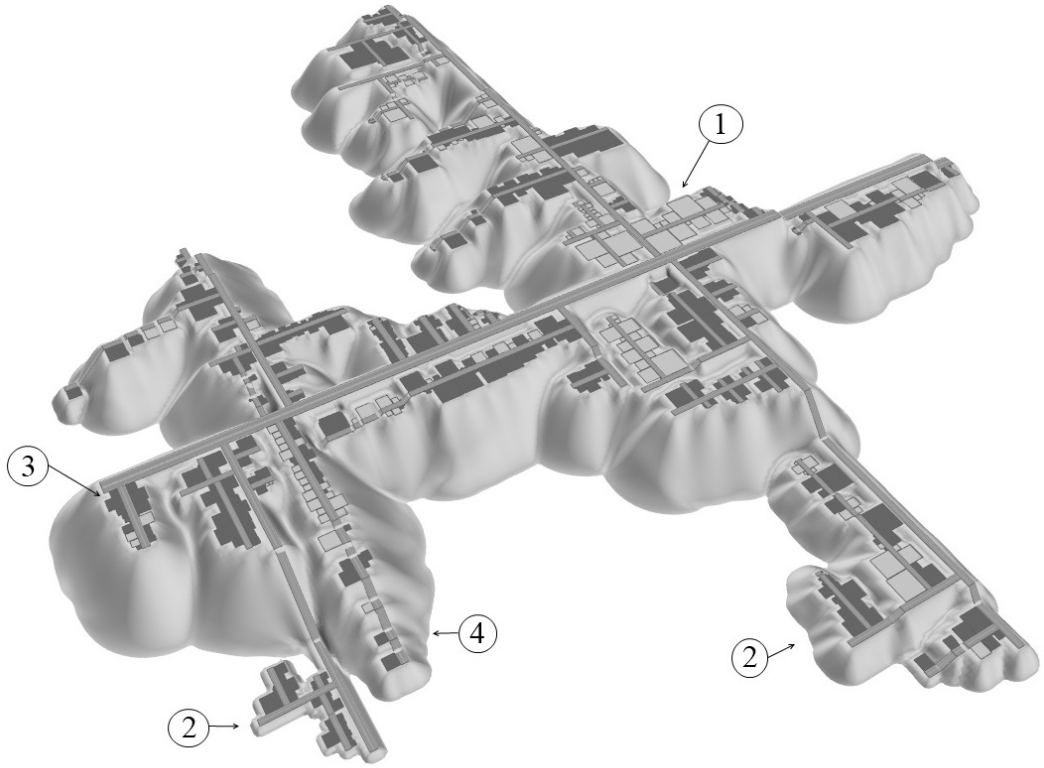


Figure 4.17: Visual Geographic Patterns

The continuity of the layouts provided by the EVOSTREETS approach, the additional depiction of the structural system evolution by means of elevation levels, and the use of terrains to improve the readability of these elevation levels allow for observing several visual patterns as shown in figure 4.17:

(1) DISUSED SITES
Disused sites represent those parts of the represented system that have been removed from the system or which migrated to other subsystems. As described in chapter 3, the software models used in this thesis capture the structure of software systems in several versions. They do not, however, include restructuring information like element renaming or the migration of elements between subsystems. Thus, our software models do not allow for distinguishing between whether a given system element is removed from the system or whether the system was restructured, i.e. the element moved to another subsystem. As a consequence, EVOSTREETS based software cities for systems which often or strongly get restructured tend to contain many of these disused sites.

(2) SUBURBS
Suburbs are cohesive software city substructures that are placed on similar elevation levels. They represent local system extensions, i.e. software elements which were added to a particular subsystem during a relatively short period of time. The larger the gap between the last addition of elements to a subsystem and the addition of the elements of the suburb, the clearer the suburb separates from the city. Finally, a suburb may become a high plateau if the corresponding software structures are not extended anymore after their initial creation.

(3) HIGH PLATEAUS
High Plateaus represent subsystems which have been part of the software for a relatively long period of time but which were not extended anymore after their initial creation. However, the corresponding software structures may actually change without effecting their high plateaus which is always the case if the content of the software elements that are represented by the tree's leaf nodes changes.

(4) MOUNTAIN SLOPES
In contrast to high plateaus, mountain slopes, i.e. structures that continuously decline on hill sides, represent subsystems that were continuously extended. The particular mountain slope (4) depicted in figure 4.17 additionally points to a subsystem that appears to be rather instable because elements were often added and removed again. A similar phenomenon is discussed by [192] on the class level (see figure 6.1(c)).

We have identified these patterns for several software systems and discuss further examples in chapter 6. While there may be more visual patterns, those listed above already show that the expressiveness of software cities can be increased significantly simply by representing system evolution in the layout. Clearly, evolutionary data can be depicted using standard means like color, shape or texture as well. But encoding evolution into the layout allows for using these standard means for other additional analysis data. Examples of such visualizations are also given in chapter 6.

4.4.4 Discussion

Temporal Resolution and Visualization Expressiveness

Increasing the temporal resolution of the software model increases the visualization expressiveness. Depicting a particular version of a system without considering prior versions yields a plain EVOSTREETS layout where no evolutionary data is included. These layouts have a low expressiveness as they depict the hierarchical system decomposition only. Adding evolutionary data, i.e. considering prior or later versions, successively increases the layout expressiveness with each version that is considered. As a result, the layouts contain successively more elevation levels representing successively more development states. The selection of these development states has a strong influence on the visual patterns appearing in the layout, e.g. suburbs can resolve and turn into mountain slopes as the temporal resolution is increased.

For monitoring scenarios the temporal resolution is defined by the monitoring process. If, for examples, the system is analyzed weekly, then the software model would be populated on a weekly basis as well. For comprehension scenarios and retrospective analyses, however, the temporal resolution can be chosen by the user. For these retrospective scenarios, however, the evolution could be analyzed to determine particular phases of the development process (e.g. phases of strong growth) to derive a corresponding temporal resolution that is compliant with these phases. Determining a best temporal resolution could increase the expressiveness of the visualizations and provide valuable insights into the development history. However, this is beyond the scope of this thesis.

Temporal Resolution and Layout Compactness

Representing the structural evolution in the layout by means of elevation as done in the EVOSTREETS approach causes a segmentation of node representations into separate, differently elevated districts. This segmentation clearly reduces the potential of packing rectangles densely in the layout and thus to obtain more compact layouts. Figure 4.18 illustrates this phenomenon for a JAVA package that is developed in four stages.

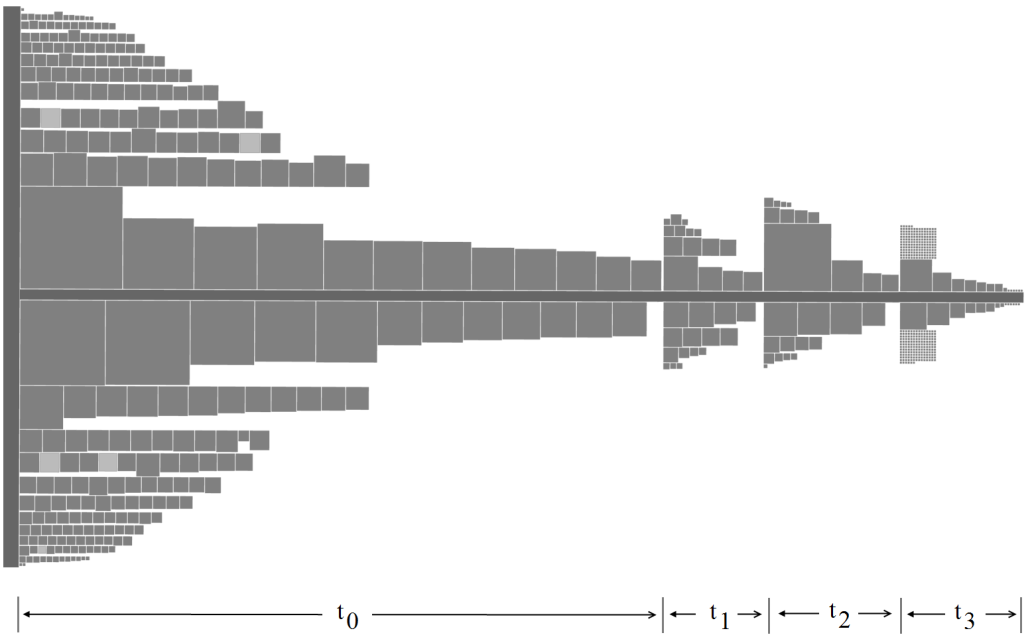


Figure 4.18: Impacts of Evolution on the Layout

Initially, the package contained a large number of classes which show up as a district t_0 that is laid out at the left hand side of the layout using a grid based rectangle packing algorithm. Later on, several classes were added in three distinct versions t_1, t_2 and t_3. These classes show up as three additional (smaller) districts that are also laid out in a grid based fashion towards the right hand side of the layout. Classes

within one district can be laid out very densely. Obviously, the compactness of the overall layout could be improved if all rectangles would be laid out into the same district which is, however, avoided by the segmentation of classes.

The effect caused by this kind of segmentation strongly depends on the temporal resolution of the software model. If the software model is populated on a daily basis, then only relatively few new elements are added to the system during this short period of time. In the worst case, a visualization is created for each new class added to the system such that all of the classes of each package are arranged sequentially. If, however, the software model is populated for each major project milestone only, then presumably much more modules will be added to the system during this long period of time. In the first case, the visualization contains many elevation levels, and thus contains relatively small districts. In the latter case, the visualization contains relatively few elevation levels and larger districts. The use of districts has a larger effect on layout compactness for coarse grained temporal resolution of the primary model. Thus, increasing the expressiveness by increasing the temporal resolution of the underlying software model at the same time decreases the layout effectiveness (with respect to layout compactness) as the potential to group buildings into compact districts is diminished.

Impacts of Restructurings and Refactorings

Assuming a sufficiently fine grained temporal resolution, frequently adding new elements to a subsystem increases the length of its street representation because new elements are attached towards the end of streets and because representations on different elevation levels cannot be grouped into districts. On the other hand, frequently removing elements from subsystems does not decrease the corresponding streets' length. Thus, even actually small subsystems with a changeful history might be represented by long streets which in turn diminish the layout compactness.

Figure 4.19: A heavily refactored JAVA Package

Figure 4.19 depicts a JAVA package that has a rather changeful history. Apparently, new classes were added and removed again over several versions. However, a manual inspection reveals another interpretation. Many of the classes that were initially part of the package or which have been added in early versions were simply renamed in the last version. These renamed classes show up as a (linear) district at the right end of the package representations. Since renaming is not detected by the SOFTWARE COCKPIT (see 3.4.1), renamed classes are interpreted as new classes and hence they are depicted twice by empty building plots depicting the renamed class and as new building plot depicting the new class under its new name.

Radial EVOSTREETS Layouts

Tauer ([174]) describes a variation of the EVOSTREETS layouts. Instead of representing the software system's root node as a central highway from which all directly contained subsystems branch off, Tauer suggests to use a central plaza as system representation. The consequence of using a circular root node representation is, however, that there is no longer a particular growing direction for the root node. A highway representation, in contrast, will always grow into one particular direction when new subsystems are added.

Winkel ([198]) derives another interesting EVOSTREETS like layout approach for software cities from so-called sunburst layouts. Again, a circular representation is used for the system's root node. However, in contrast to [174], Winkel uses an implicit representation of the containment relation instead of indicating containment explicitly by city elements like streets. Unfortunately, a considerable shortcoming of this approach is the large amount of unused space that is typically left in the center of the city representation.

Radial variants are beyond the scope of this thesis. In the remainder of this work, we will focus on orthogonal EVOSTREETS variants as discussed above.

4.5 Summary

Important quality criteria for software cities are visualization expressiveness, effectiveness, and consistency. Each of these visualization criteria is influenced by corresponding layout criteria. In this chapter, we reviewed currently used layout approaches and discussed their consistency, expressiveness, and effectiveness properties. From this discussion the EVOSTREETS layout approach has been derived. The EVOSTREETS approach uses a continuous arrangement of nodes and preserves the input order. Both characteristics combined allow for a very specific mapping of evolutionary data: Representing the age of software elements by the elevation of their corresponding buildings yields software cities underpinned by coherent elevation landscapes forming expressive visual patterns for particular evolutionary phenomena. The EVOSTREETS approach thus offers a higher expressiveness than the other approaches used for software cities so far because in addition to system decomposition it represents the system evolution in the layouts.

We have also pointed out that the EVOSTREETS approaches presumably yield highly consistent layouts for evolving software cities because they are based on a treemap approach (Slice&Dice) which has already proven to be less susceptible to change. On the other hand, compared to other approaches, EVOSTREETS layouts may suffer from lower effectiveness caused by an inefficient use of the layout space. Both properties, layout consistency and layout effectiveness, are analyzed in the next chapter.

The EVOSTREETS approach is based on a simple observation: By relaxing the space-fillingness characteristic, treemaps can easily be turned into a corresponding rectangle packing. The major disadvantage of treemaps, i.e. their variable and possibly

very high aspect ratios, can thus easily be avoided while their consistency characteristics may be maintained. Previous research has already shown the high consistency of Slice&Dice treemaps. Thus we derived our EVOSTREETS approach from the Slice&Dice approach. To reduce the inefficient use of layout space caused by long streets we modified the approach to lay out nodes to both sides of their parent representation and by using optional packing algorithms. From this particular perspective the (linear) EVOSTREETS approach is a double-sided Slice&Dice packing algorithm which could easily be modified to produce double-sided Slice&Dice treemaps.

Science is built up of facts, as a house is built of stones; but an accumulation of facts is no more a science than a heap of stones is a house.

Henri Poincaré
(1854 - 1912)

Chapter 5

Analysis of Layout Properties

Effective, expressive, and consistent Software Cities require the use of expressive, effective, and consistent layouts. As discussed in the previous chapter, not all of these criteria are simultaneously fulfilled by the layout approaches currently used for software cities. Therefore, the EVOSTREETS approach has been designed to provide an expressive, effective, and consistent platform for software cities. In this chapter, all layout approaches discussed in chapter 4 are compared with respect to these quality criteria.

In the first section, the goal of this chapter and the analysis methods are clarified, and data upon which the analyses are performed are defined. In sections two and three, the layout quality criteria *layout consistency* and *layout effectiveness* are analyzed. Finally, the results of both analyses are summarized.

5.1 Goals, Methods, Data Sets

For software quality analysis of a particular system state or for retrospective analysis of software evolution consistency may be less important than effectiveness and expressiveness. In the former case consistency is even irrelevant, whereas in the latter case it can easily be achieved using a supergraph visualization. In contrast, when monitoring ongoing software development consistency may become much more important than effectiveness or expressiveness. The goal of this chapter is thus to characterize the different layout approaches with respect to the aforementioned criteria.

Regarding layout expressiveness, the approaches are well understood: All depict the hierarchical system decomposition and a particular leaf node property that is typically mapped onto rectangle size. Treemaps additionally represent an aggregate value of that property, whereas the EVOSTREETS approach additionally depicts the system evolution. Expressiveness is not discussed further. Instead, this chap-

ter concentrates on layout consistency and layout effectiveness. Both qualities are evaluated separately in subsections 5.2 and 5.3, respectively. The evaluation of the characteristics of the different layout approaches is done empirically by measuring the layout consistency and the layout effectiveness for several real world (open source) software systems. Sections 5.1.1 and 5.1.2 list the layout approaches that are analyzed and the software systems that are used for this evaluation. The measures that are used for the evaluation of layout consistency and layout effectiveness are described in the corresponding subsections.

Mannl ([132]) previously analyzed the stability and compactness of three layout approaches. The analyses described in the remainder of this chapter differ from the analyses described in [132] in several ways. First, we analyze different layout approaches. We exlcude force-directed approaches, but include several variants of the EVOSTREETS and Treemap approaches. Also, we include additional methods like the DISTRIBUTION analysis to determine a characteristic performance of each layout approach and the SUSCEPTIBILITY TO SMALL CHANGES analysis to determine the impact of particularly small changes on layout consistency. Beyond these differences, we also give an overview of current similarity measures and select the measures to be used for our analyses on the basis of an empirical evaluation described in the literature ([35]).

5.1.1 Layout Approaches

The following layout approaches are analyzed:

E_L: EVOSTREETS, Linear. Linear indicates that leaf nodes are arranged linearly along their ancestor's line representation. Leaf nodes are sorted by initial size. The sorting is maintained for all versions.

E_P: EVOSTREETS, Packing. Packing indicates that leaf nodes are arranged in districts using the packing algorithm described in [192]. The packing algorithm includes re-sorting the input list.

E_G: EVOSTREETS, Grid. Grid indicates that leaf nodes are arranged in a grid fashion similar to the Strip Treemap approach. Leaf nodes are sorted by initial size. The sorting is maintained for all versions.

P: Packing. The rectangle packing algorithm used in the CodeCity approach ([192]). The algorithm re-sorts the input list for each version and uses a discontinuous layout strategy.

T_{SD}: Slice&Dice Treemap as discussed in section 4.2.1.

T_{St}: Modified version of the Strip Treemap algorithm discussed in section 4.2.1. The algorithm used here alternates the layout direction between each two successive strips. The modification avoids discontinuities; hence it potentially improves layout consistency, but has no further impacts on expressiveness and effectiveness.

T_{Sq}: Squarified Treemap as discussed in section 4.2.1.

System	Removed Versions
Apache Ant	8
ArgoUML	16, 13
Compass	13
Datanucleus Core	30
Hibernate Core	29
CrocoCosmos	42, 18

Table 5.1: Removed Revisions

T_{P_*}: Pivot Treemaps as discussed in section 4.2.1. We consider all of the pivot selection strategies, i.e. Pivot-By-Middle T_{P_M}, Pivot-By-Size T_{P_S}, and Pivot-By-Split-Size $T_{P_{SS}}$.

Three EVOSTREETS variants are considered. E_L is included because it promises a high consistency which possibly comes at the price of a lower effectiveness (i.e. worse compactness). Differences can be expected between E_P and E_G which both target a higher layout compactness. E_P uses the rectangle packing algorithm used in the CodeCity approach, thus it includes re-sorting and may suffer from discontinuities. E_G may be a compromise between E_L and E_P since it uses a more compact layout than E_L but avoids large discontinuities that may occur for E_P.

Besides the EVOSTREETS variants, the evaluation also includes the packing algorithm used in the CodeCity approach and most treemaps discussed in chapter 4. Spiral treemaps are not considered because a similar modified Strip Treemap is included. Whereas Spiral Treemaps arrange strips in a spiral manner, the modified Strip Treemap arranges them horizontally in alternating layout direction. Both approaches thus avoid discontinuities.

5.1.2 System Selection and Preparation

Layout consistency and layout effectiveness are analyzed for a set of open source software systems taken from several freely accessible repositories. The systems had to be of reasonable size and provide a sufficiently large evolution history. Surely, any kind of size limit is arbitrary and thus subject to criticism, but (for the purpose of this work) we consider systems with at least several hundred classes in their respective last available version as sufficiently large.

If available, official releases are analyzed, otherwise versions are taken from the respective version control system on a regular pattern, e.g. every 100$^{\text{th}}$ commit. For each two successive versions for which we could not identify any structural change, the latter version was removed from the data set. Apparently, identical versions may occur when e.g. changes were applied outside the observed package or repository scope, or if some other resources than the source code were changed from one version to the next. Table 5.1 lists those versions. Furthermore, only those systems are considered for which at least 16 versions remain after cleaning the version set. For quartile based analyses and a uniform distribution of values this yields a set of

System	# Versions	# Classes (initial)	# Classes (final)
Apache Ant	21	94	1044
Apache CXF	42	1490	2760
ArgoUML	16	908	1921
Checkstyle	26	23	692
Compass	19	1174	1539
CrocoCosmos	49	389	509
Datanucleous Core	47	612	849
FindBugs	22	67	1024
Hibernate.Core	42	1374	2696
JFreeChart	51	90	611
JME3	19	736	898
JMol	22	542	748
Mule.Core	43	564	1131
Neo4J	26	364	561
ProcessDash	37	55	925
Spring Framework	49	708	2150
	$\sum$ 531		

Table 5.2: Software Systems for Analysis

four data points for each quartile.

An overview of all software systems that are considered somewhere in this thesis including the corresponding repositories and available releases or versions is provided in appendix A. Table 5.2 lists those systems that are used for the analysis of layout consistency and layout effectiveness in this chapter. All systems are written in JAVA. Their initial sizes range from a few tens to several hundreds of classes. Their final sizes range from a few hundred to several thousand classes. The list contains systems which experienced an enormous increase in size (e.g. Checkstyle, Findbugs, ProcessDash) as well as systems which increased only by a small factor. They cover the typical system sizes addressed in this thesis. Altogether these 16 systems provide 531 versions and thus 515 transitions between successive versions.

5.2 Layout Consistency

In this section, the consistency of the layout approaches listed in 5.1.1 is analyzed. We first further refine and clarify the analysis goals in subsection 5.2.1 and describe the analyses that are performed later in this chapter. These analyses are based on several measures to quantify the degree of similarity between each two layouts. These measures are discussed in subsection 5.2.2. Subsection 5.2.3 describes how to apply these measures to allow for a comparison of different layout approaches. The results of the consistency analyses are discussed in subsection 5.2.4.

5.2.1 Analysis Goals and Method

The goal of this section is characterize the suitability of different layout approaches for application during ongoing system development. Approaches which frequently yield inconsistent layouts even if the underlying software systems do not change much are only of limited use for ongoing software development. The analysis thus addresses the following questions:

- Which layout approaches perform best? How large are the differences between the approaches? Ideally, we can determine a ranking of the approaches.
- Do small software changes cause large layout adaptations? Which approaches are most susceptible to small changes?
- The additional use of rectangle packing algorithms in the EVOSTREETS approach targets more compact layouts. Does the use of such algorithms (on the other hand) reduce the consistency of the EVOSTREETS layout?

Layout inconsistencies may be caused by changes of various rather different layout aspects like absolute positioning of nodes, the node clusters, or the global and local layout structure. Consequently, layout consistency can hardly be determined using only one particular measure. Instead, it must be characterized by a set of measures that in their entirety cover the most relevant of these aspects. Therefore, we select three established measures from the literature that each covers a particular similarity aspect: node positions, node neighborships, and node orderings. The selection of the particular measures is discussed in section 5.2.2.

Applying these three measures to 515 layout transitions for 10 different layout approaches (listed in section 5.1.1) yields a total of 15.450 data points. To answer the aforementioned analysis questions, these data points must appropriately be aggregated and analyzed. To get a comprehensive characterization of consistency we use three different analyses. Each of them is performed separately for every similarity measure:

- TOP-PERFORMER
  For each measure one can easily count how often each layout approach performs best over all layout transitions. However, such counting is rather restrictive since there might be a dominant high performer such that a slightly worse performance of other approaches would not be detected. For this reason, we additionally count how often an approach is among the top three performers. A layout is among the top three iff no more than two other approaches perform better, i.e. there may be more than three top-three performers.
- DISTRIBUTION ANALYSIS
  Besides counting the frequencies of high performance, we also analyze the distribution of the measurement values for each approach separately. This analysis is based on box-plot representations (as described in appendix B, p. 195) that depict the minimum, maximum, average, and median values as well as the interquartile range for each measure and each system separately. As a

result, this analysis allows for characterizing the typical behavior of a given approach with respect to the respective similarity measure.

- SUSCEPTIBILITY TO SMALL CHANGES
  Consistency is highly important if the layout is used with a high frequency (e.g. on a daily basis) because users may still be familiar with the previous layout. For this reason, we analyze the impact of small software changes on layout consistency. Software changes are quantified using a simple measure called *Relative Weight Change*, RWC_w, which is defined for dynamic hierarchical attributed graphs $DHAG = (G, T, R, f_e, A)$ (as discussed in section 3.3.2) as

$$RWC_w(DHAG, t) := \frac{\sum_{\{v \in G | f_e(v,t)=True\}} |a_w(v,t) - a_w(v,t-1)|}{\sum_{\{v \in G | f_e(v,t)=True\}} a_w(v,t-1)}$$

  with $a_w \in A$ denoting a particular attribute function that is used to determine the size of node representations in the layout. The RWC measure thus quantifies the degree to which the occupied layout space changes between two successive versions. Its values cover R_0^+. Using a plot based representation we analyze how the RWC measure and particular layout similarity measures correlate. Since we are particularly interested into small software changes, we considered only transitions with an RWC value of at most 0.5. For the software systems listed in table 5.2, this filtering step drops 23.49% of all layout transitions.

The similarity measures for which each of these three analyses is performed are discussed in the next section.

5.2.2 Layout Similarity Measures

We determine layout consistency on the basis of layout similarity measures. In this section we discuss the requirements that these similarity measures must fulfill in the context of this work. We describe and discuss well known similarity measures for graph layouts that can be found in the literature.

Requirements

LAYOUT BASED SIMILARITY
Layouts are basically assignments of positions to nodes. Once layouts are computed, corresponding images depicting these layouts can be computed as well. These images can now be compared with corresponding images for previous versions. Similarity would be determined on an image level which, however, would not take into account the semantic differences in these images. Seemingly identical images can depict different data. For example, two visually identical objects simply might change their positions which could not be detected. Consequently, similarity must be computed on the basis of layout rather than its visualization.

COMPARABILITY OF DIFFERENT LAYOUT APPROACHES
The layout approaches that are compared with each other make use of very different visual clues. On the one hand, treemaps and packing approaches use nested areas (city districts) whereas the EVOSTREETS approaches, on the other hand, use linear representations (streets). Since district offsets and street widths can be set individually, the effectiveness of these clues depends on user settings. For this reason, the measures should not account for these visual clues. Besides, different approaches use different layout scales: While treemaps are always rendered into the same bounding box, rectangle packing and EVOSTREETS based layouts grow beyond such unit squares. Therefore, normalization must be used to assure the comparability of the similarity values.

INVARIANCE AGAINST GEOMETRIC TRANSFORMATIONS
Invariance against simple geometric transformations like rotation, linear scaling, and translation is an often stated requirement for similarity metrics because these transformations can easily be performed by users either mentally or (given appropriate tool support) interactively. Rotation and translation are not relevant in the context of this work because the layouts are always laid out into the same direction beginning at the coordinate origin. As discussed above, the approaches may use different scales, but scaling is also relevant if layouts produced by the same approach evolve, i.e. if they grow larger or become smaller. Thus, again, normalization is necessary for some measures. We will discuss this point in further detail later.

Related Work

The following layout based measures to determine similarity between layouts are discussed in the literature. These measures analyze different aspects that have an impact on layout similarity and, consequently, on layout consistency. The aspects can roughly be divided into node displacements, changes in node neighborships/proximities, and changes of the spatial ordering of nodes in successive layouts. We group the measures into these categories, but other groupings are discussed as well (e.g. [139] group into orthogonal ordering, proximity, and topology, or [144] group into geometry and topology).

MEASURES FOR NODE DISPLACEMENTS
Measures based on node displacements compare node positions between successive versions. The idea behind all measures described below is that it is easier to regain familiarity if nodes stay close to their original position between successive versions.

- Euclidean Mental Distance ([58])
  Diehl et al. describe a simple measure called *Euclidean Mental Distance* $\Delta_{\|}$ which accumulates absolute node movements between successive layouts. For two layouts l_1 and l_2, $\Delta_{\|}$ is defined as

  $$\Delta_{\|}(l_1, l_2) = \sum_{v \in V_1 \cap V_2} dist(l_1(v), l_2(v))$$

  with $dist(l_1(v), l_2(v))$ yielding the euclidean distance between the node position $l_1(v)$ of v in l_1 and the node position $l_2(v)$ of v in l_2. The measure has not

been validated. It is not invariant against a variety of geometric transformations including layout scaling. Its values in general increase with both, layout size and number of nodes in the graph.

- Average Displacement of Nodes ([71])
  Several authors use measures similar to the Euclidean Mental Distance, but normalize the measurement value by the number of nodes in the graph. This normalization yields an *average* displacement of nodes. Besides *average displacement of nodes* [71], this measure is also called *average distance change* [23], or simply *average distance* [35].

- Distances Moved Model ([127])
  For two layouts l_1, l_2, the Distances Moved Model DM is defined as

$$DM(l_1, l_2) = \frac{\Delta_{\|}(l_1, l_2)}{UB}$$

  with $\Delta_{\|}(l_1, l_2)$ as defined above. In contrast to the aforementioned measures ([58], [71]), the accumulated node displacements are normalized by an upper bound UB: In layouts that fit the unit square nodes move at most $\sqrt{2}$ units. Hence, $UB = n\sqrt{2}$ and the measure is actually normalized by the number of graph nodes n.

- Layout Distance Change ([23])
  [23] measure *change* for evolving treemap layouts. The authors propose the Layout Distance Change measure which is based on a distance function $d(r_1, r_2)$ that determines a distance between two rectangles r_1 and r_2:

$$d(r_1, r_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (w_1 - w_2)^2 + (h_1 - h_2)^2)}$$

  x, y, w, and h denote the corresponding rectangles' coordinates, their width, and their height, respectively. On the basis of this distance function, the *layout distance change* measure is is defined for two layouts as the average distance (with respect to the aforementioned distance function) between all corresponding rectangles of both layouts. In contrast to the previous measure, the layout distance change measure also considers changes in width and height.

- Nearest Neighbor Between ([35])
  The *Nearest Neighbor Between* measure is based on the assumption that users can memorize node positions and thus expect each node to be placed closest to its original position. The measure counts the number of nodes for which this requirement is not fulfilled, i.e. those nodes for which (in the subsequent layout) another node is placed more closely to the node's original position.

- Average Relative Distance ([34])
  In contrast to the average displacement of nodes (([71])), the average relative distance $rDist$ determines the average change of distances between each two nodes for successive layouts l_1 and l_2. $rDist$ is defined as

$$rDist(l_1, l_2) = \frac{1}{n^2 - n} \sum_{v,w \in V} |dist(l_1(v), l_1(w)) - dist(l_2(v), l_2(w))|$$

with $dist(p, q)$ yielding the euclidean distance between the node positions p and q. $l_1(v)$ and $l_2(v)$ return the position of node v in the layout l_1 and l_2, respectively.

- Hausdorff Distances
  Hausdorff Distance based measures determine a degree of match between two point sets. Several such measures are discussed in the literature, [59] give a systematic overview and validate their performance. One of them (the *Modified Hausdorff Distance*) is used by Kuhn et al. ([109]) to determine the stability of thematic software maps. In general, measures based on the Hausdorff Distance do not take node identities into account. Consequently, these measures may return low values even for very inconsistent layouts (e.g. layout where randomly two nodes change their positions). This characteristic makes them less suitable for measuring layout consistency.

MEASURES FOR NODE NEIGHBORSHIPS

Besides node positions and displacements, node proximities and neighborships are another characteristic that potentially influences the consistency of layouts. Measures in this group do not consider the magnitudes of node displacements. Instead, they typically determine how distances between closely placed nodes change.

- Nearest Neighbor Within ([35])
  The Nearest Neighbor Within measure determines the number of nodes whose nearest neighbor changes between successive versions. Nearness is typically based on the euclidean distance. The result is normalized by the number of nodes in the layout which yields a range of values from 0 to 1.

- ϵ-clustering ([62])
  The idea of ϵ-clusters as described in [62] is that if a layout contains clusters of nodes, then these clusters should be maintained during evolution. An ϵ-cluster of a node n includes all nodes whose distance to n is less than some defined ϵ value. For a given set V of nodes, the authors suggest to choose

  $$\epsilon = \max_{v \in V} \min_{w \in V, v \neq w} dist(v, w)$$

  which is the maximum of all minimum distances of each node to any other node in the layout. Changes of ϵ-clusters can be quantified by counting how many nodes enter or leave ϵ-clusters of other nodes ([35]). ϵ-clusters must be used with care: If ϵ is chosen separately for two successive layouts, then ϵ-clusters are susceptible to outliers, otherwise they are susceptible to layout scaling.

MEASURES FOR NODE ORDERING

Measures in this group are based on the spatial arrangement of nodes in the plane and quantify changes hereof. Neither do they depend on displacement magnitudes of nodes, nor on proximities between nodes.

- Orthogonal Mental Distance ([58])
  [139] discuss the importance of preserving the horizontal and vertical ordering of nodes in successive layouts. The *Orthogonal Mental Distance* $\Delta_\perp$ ([58]) determines changes of this particular kind of ordering. $\Delta_\perp$ is defined as

$$\Delta_\perp(l_1, l_2) = \sum_{v_1,v_2 \in V_1 \cap V_2} (|sgn(l_1(v_1).x - l_1(v_2).x) - sgn(l_2(v_1).x - l_2(v_2).x)| + |sgn(l_1(v_1).y - l_1(v_2).y) - sgn(l_2(v_1).y - l_2(v_2).y)|)$$

  $sgn(v) \in \{-1, 0, 1\}$ yields the sign of a real number v. $l_1(v_1).x$ denotes the x coordinate of node v_1 in the layout l_1. $\Delta_\perp$ counts the number of pairs of nodes whose orthogonal order is not preserved in successive layouts.

- Ranking ([35])
  Similarly to the Orthogonal Mental Distance $\Delta_\perp$, the *Ranking* measure described in [35] quantifies changes of the orthogonal arrangement of nodes. The measure is (in contrast to $\Delta_\perp$) normalized by the number of nodes in the layout which allows for comparing differently sized layouts with each other. For two layout l_1 and l_2, rnk is defined as

$$rnk(l_1, l_2) = \frac{1}{UB} \sum_{v \in V} min\{|rg(l_1(v)) - rg(l_2(v))| + |abv(l_1(v)) - abv(l_2(v))|, UB\}$$

  The upper bound UB is defined as $UB = 1.5(|V| - 1)$. $rg(p)$ returns all nodes on the right hand side of p, i.e. nodes whose x coordinate is larger than the x coordinate of p. Similarly, $abv(p)$ returns all nodes above p, i.e. nodes whose y coordinate is larger than the y coordinate of p.

- Angular Change ([34])
  The *Angular Change* measure ([34]) accumulates all angular changes between each two nodes. An interesting characteristic of this measure is that it provides a built-in weighting of node movements: Displacements at far distances yield lower values than the same displacements at close distances. On the other hand, the angular change is sensitive to the direction of node displacements, e.g. displacements of nodes which move directly towards each other are not detected.

- λ-Matrix [127]
  The λ-Matrix is based on the assumption that nodes may be used for orientation. For each two nodes (that potentially serve as landmarks) a user may remember the side on which a third node is placed. Thus, the measure determines for each pair of nodes the number of nodes that cross the connecting direct line between them.

The measures described above cover several different aspects, i.e. node displacements, node neighborships, and node orderings. Clearly, the consistency of evolving layouts is reduced if these characteristics change during evolution. Besides these,

more characteristics are discussed in the literature as well but they are less relevant in the context of this work because they relate to e.g. edge representations ([34]). We concentrate on the aforementioned characteristics but do not use all of the discussed measures.

Measure Selection

Not all of the measures described above are validated or have been shown to correlate well with how users judge or estimate layout similarity. [35] performed a study and validated several of the aforementioned similarity measures. For each group of similarity measures (displacements, neighborships, and node ordering), we choose this similarity measure that performed best in this evaluation. The measures are:

- Displacements: Average Displacement of Nodes (ADN)
- Neighborships: Nearest Neighbor Within (NNW)
- Node Ordering: Ranking

Except for the *Layout Distance Change* measure ([23]), all measures are point based, i.e. they do not take node dimensions into account. Hence, for the analyses performed in the remainder of this section, we ignore the width and the height of node representations. Instead, we consider their middle points even though these representations are two dimensional. Clearly, this reduction has effects. Concerning displacements, the same node displacement might be regarded more disturbing for small rectangles than for larger ones. Concerning neighborships, spatially adjacent large rectangles might be considered closer to each other than smaller and spatially not adjacent rectangles even if the distances between their respective middle points are equal in both cases. Concerning ordering, rectangles define a much broader space of potential node arrangements than points which allow for comparisons with respect to quadrants, only. However, these effects have limited influence on the similarity values because they decrease with increasing distance between nodes.

Normalization

Different layout approaches may use different layout sizes. Whereas treemaps typically are drawn into a predefined area (e.g. a unit square), the size of rectangle packing as well as EVOSTREETS based layouts is not known in advance but the result of the layout process. Different layout sizes do not affect the NNW and $Ranking$ measure, but they have a large influence on the ADN measure: A node displacement in a treemap layout that is drawn into a unit square in general is not comparable with the same absolute node displacement in a much smaller or larger rectangle packing or EVOSTREETS layout. To allow for a comparison of these approaches with respect to the ADN measure, layouts must be normalized by fitting them into a standard rectangle before applying the similarity measures.

Different layout sizes may also result from system evolution: If the represented software system grows, the layout might grow as well. In such cases, normalization

may cause node displacements even for those structures that did not change their absolute positions from one version to the next. To minimize these effects, a computation of a best match displacement of the entire layout is surely possible. On the other hand, a normalization that simply fits a layout into a standard rectangle more appropriately reflects the intended application scenarios, e.g. if the user fits a layout to the screen. For this reason, a best match analysis is not used in this work.

5.2.3 Experiment Setup and Execution

Layout Settings

District offsets and street widths can be adjusted by users. To avoid an influence of these individual settings on the analysis of layout consistency, the following settings are used for the analysis:

- For all EVOSTREETS approaches, the width of inner tree nodes' line representations (streets) is set to zero. No offsets between rectangles are used.
- For all EVOSTREETS approaches, elevation levels are not used which is achieved by setting the elevation level increment to zero. Elevation levels may cause additional node displacements and thus induce more inconsistencies to evolving layouts. On the other hand, they are an optional feature of the EVOSTREETS approaches, and their actual magnitudes depend on user preferences. Therefore, they are not regarded during this analysis.
- For all treemap approaches and the rectangle packing approach, offsets are set to zero.
- All treemaps are drawn into a 1×1 square rectangle.

The Node Set

The graphs in this thesis are dynamic: New nodes appear, others disappear, node properties change. Clearly, each graph version can be laid out separately and the resulting layouts can be compared with the previous ones. In this case, the previous and current node sets intersect and the analysis must be performed on the intersecting node set of both graph versions.

In the EVOSTREETS approaches deleted system structures remain represented in the software model as well as in the spatial software model. For reasons of comparability, these deleted structures must also be represented in the other layout approaches, i.e. the treemap approaches and the rectangle packing approach. Therefore, all layouts are computed on the basis of the supergraph which contains deleted and existing nodes. Consequently, the node set typically grows. Deleted nodes are laid out with respect to their last spatial dimensions (width and height), i.e. removed structures do not change their appearance.

Local vs. Global Application

The graphs used in this thesis are hierarchically structured. The treemap, rectangle packing, and EVOSTREETS algorithms make use of this characteristic by recursively laying out hierarchical substructures as visual sub-layouts for non-leaf nodes. Adaptations like a displacement of a container rectangle (or city district, respectively) may be easier to recognize and understand on this coarse-grained level than several smaller displacements of leaf nodes even if the corresponding similarity measure would yield similar or worse values.

The similarity measures described above do not account for this aspect since they were designed for graphs which have no hierarchy defined on their nodes. The hierarchical graph structure can, however, be respected during measurement by restricting the node set to which the measures are applied, a distinction which is called *local* and *global* application in this thesis. Depending on the type of measure, this restriction takes different forms:

- Binary Measures
  Measures which quantify changes of a characteristic that concerns two nodes (like the distance between them, or their spatial arrangement) can be restricted to sibling nodes only, i.e. nodes which have the same direct ancestor in the hierarchy tree.

- Unary Measures
  Measures which quantify changes of a characteristic that concerns only single nodes (like node displacement measures) can be restricted to quantify changes *relative* to the direct ancestor in the hierarchy tree. An example is described in [4] where node displacements are computed relative to the center of the cluster they belong to rather than the center of the overall graph layout. Cluster movements thus have no effects on the measurement values for single nodes of the cluster.

Clearly, hierarchical structures are important visual clues, but applying measures both locally and globally raises more questions than it provides answers for: Restricting the computations to sibling leaf nodes of the same parent yields only very few results which in turn rises the question of how to include inner, hierarchical nodes, particularly which nodes and pairs of nodes to consider and how to weight their measurement values. For this reason, we apply the measures only globally on the leaf node level without considering the hierarchy tree.

Procedure

The procedure to obtain and prepare the approximately 15000 data points for analysis is straightforward: Each system is separately loaded into the visualization tool which iterates over all versions and at each step computes normalized layouts for all approaches (EVOSTREETS, Treemaps, Packing). As discussed above, normalization has no effects on NNW and $Ranking$ but avoids incomparable results for the ADN measure. The layouts are stored in memory such that all measures can be computed

between each two successive versions. The results of all measurement computations are stored persistently into result files. After this procedure has been applied to all systems, a separate analysis tool imports the result data and automatically generates the corresponding tables and diagrams shown in the remainder of this chapter.

5.2.4 Results and Discussion

In this section, the results for the analysis of consistency are presented. As described in 5.2.1, each measure (ADN, NNW, $Ranking$) is discussed separately but includes the same set of analyses, i.e. analysis of Top-Performers, analysis of metric distribution, and analysis of susceptibility to small changes. The results of all analyses are summarized for each measure. The results of each measure are compared with each other in a closing discussion at the end of this section.

Average Displacement of Nodes (ADN)

TOP-PERFORMER
Tables 5.3 and 5.4 show the results of the Top-Performer analysis for the ADN measure. The following observation can be made:

- T_{SD} performs best for 180 layout transitions (table 5.3), followed by E_L (154 transitions), and E_G (124 transitions). E_P clearly falls behind E_G, E_L, and T_{SD}. However, it still performs better than the remaining treemap approaches and P.

- Results are similar concerning the top three performances shown in table 5.4: T_{SD} is among the top three performers in 386 cases, followed by E_L and E_G. E_P falls behind E_G, E_L, and T_{SD} but still performs better than the remaining treemap approaches and P. P performs worst as it reaches the top three positions three times, only.

- P, T_{P_M}, T_{P_S}, and T_{St} occasionally perform best. $T_{P_{SS}}$ and T_{Sq} perform worse than T_{P_M}, T_{P_S}, and T_{St} with respect to both, best performance and top-three ranking. P performs worse than all other approaches.

- It is always an EVOSTREETS variant or the Slice&Dice treemap that performs best for each system. Among the EVOSTREETS variants, E_L performs better than E_G and E_P.

For 49 transitions E_P performs best and is among the top three performers for 187 transitions, i.e. in 36% of all transitions. This relatively large difference for compared to the other approaches (particularly the E_G approach) indicates the impact of combining the EVOSTREETS with a re-sorting and discontinuous layout strategy.

P performs surprisingly bad. For two transitions only, it performs best, for one more transition it is among the top three performers. In both cases, the change of

System	E_L	E_G	E_P	T_{SD}	T_{St}	T_{Sq}	T_{P_M}	T_{P_S}	$T_{P_{SS}}$	P
Apache Ant	6	6	0	8	0	0	0	0	0	0
Apache CXF	13	8	4	16	0	0	0	0	0	0
ArgoUML	2	1	2	9	0	0	0	1	0	0
Compass	8	3	0	7	0	0	0	0	0	0
Datanucleus Core	19	13	1	12	0	0	1	0	0	0
Hibernate Core	5	9	0	25	1	0	0	0	0	1
JFreeChart	24	14	2	11	0	0	0	0	0	1
Mule Core	13	5	7	17	0	0	0	0	0	0
Neo4J	8	5	0	12	0	0	0	0	0	0
Spring Framework	11	12	6	19	0	0	0	0	0	0
JME 3	4	5	1	8	0	0	0	0	0	0
JMol	3	6	1	8	2	0	0	1	0	0
Checkstyle	2	9	6	8	0	0	0	0	0	0
CrocoCosmo	16	15	8	8	0	0	1	0	0	0
FindBugs	13	8	0	0	0	0	0	0	0	0
ProcessDash	7	5	11	12	0	0	1	0	0	0
$\sum$	154	124	49	180	3	0	3	2	0	2

Table 5.3: Top Rankings for ADN

System	E_L	E_G	E_P	T_{SD}	T_{St}	T_{Sq}	T_{P_M}	T_{P_S}	$T_{P_{SS}}$	P
Apache Ant	19	17	3	18	2	0	1	0	0	0
Apache CXF	27	25	10	33	10	2	6	8	2	0
ArgoUML	10	11	7	13	2	0	0	2	0	0
Compass	15	10	5	13	4	0	6	1	0	0
Datanucleus Core	37	30	8	37	9	4	7	4	2	0
Hibernate Core	22	21	8	37	10	3	10	7	4	1
JFreeChart	44	44	17	35	1	0	5	2	1	1
Mule Core	30	28	18	32	8	0	6	3	1	0
Neo4J	21	19	2	24	4	3	1	1	0	0
Spring Framework	34	34	27	36	6	0	3	4	0	0
JME 3	10	9	7	15	6	2	4	1	0	0
JMol	7	12	4	18	5	4	7	5	1	0
Checkstyle	24	25	14	11	0	0	1	0	0	0
CrocoCosmo	32	32	25	31	8	2	8	3	3	0
FindBugs	21	21	11	10	0	0	0	0	0	0
ProcessDash	25	22	21	23	2	3	4	5	2	1
$\sum$	378	360	187	386	77	23	69	46	16	3

Table 5.4: Top 3 Rankings for ADN

node weights is several magnitudes lower than for the other transitions. The actual changes applied to the corresponding software system are extremely small and can hardly be identified in the visualization at all. The ADN values for all approaches are similarly low for this transition such that these occurrences rather happened incidentally.

For only three layout transitions the Strip treemap approach performs best, and for only 77 transitions (i.e. for only 14% of all transitions) it reaches a top three ranking. This is a surprisingly low performance because the Strip treemap approach uses a continuous layout strategy as well.

DISTRIBUTION ANALYSIS

A complete set of diagrams for all 16 software systems is provided in the appendix B, pp. 199(ff). The magnitudes covered by the distribution boxplots vary strongly among systems because each system is visualized with a particular temporal resolution. Releases for one system may contain much more or less change relative to their size than releases of other systems. Despite these differences, some patterns appear:

- For all systems, P has the highest median and the highest average value. For the majority of systems, its interquartile range covers larger values than all other approaches. Only occasionally, other approaches cover comparably large values. In these cases, however, the median and average ADN values are larger for P.
- E_L, E_G, E_P, and T_{SD} have lowest median and average values and a narrow interquartile range compared to the other approaches. Though in some cases, other approaches show similar characteristics. If so, these approaches have higher maximum values and/or a larger interquartile range.
- The diagrams do not show a clear trend towards one of the EVOSTREETS variants.

The interquartile range covers 50% of all nodes. For P, the interquartile range covers rather large ADN values in comparison to the other approaches, which indicates that relatively large node displacements are a typical characteristic for P.

Concerning the EVOSTREETS variants, all distribution characteristics are low compared to most of the other approaches. This is interesting as it weakens the worse performance of the E_P variant shown in the top-performer analysis. Though not very often among the top three, E_P performs rather well as its absolute values are close to E_L and E_G.

SUSCEPTIBILITY TO SMALL CHANGES

Susceptibility to small Changes is analyzed using a plot representation depicting the ADN values for all layout transitions on the y axis in dependence of the actually applied software changes on the x axis. Figures 5.1 and 5.2 show the plots for P and E_L. A complete set of diagrams for all approaches is provided in appendix B. The following observations can be made:

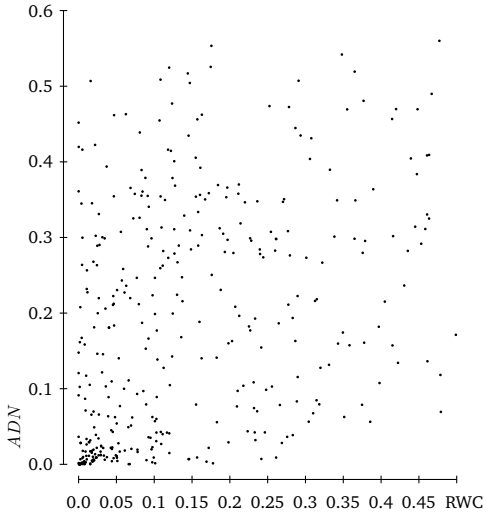


Figure 5.1: ADN Plot for P

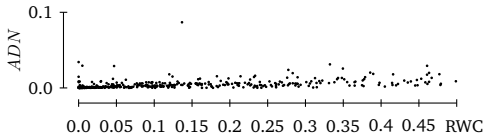


Figure 5.2: ADN Plot for E_L

- Compared to the other approaches, the ADN values for E_L, E_G, E_P, and T_{SD} are relatively low. Only very few transitions cause ADN values close to 0.1. In the majority of cases, the values are clearly below 0.1. At this scale, there are no noticeable differences between E_L, E_G, E_P, and T_{SD}.

- P yields the highest values of all approaches. There is a slight concentration of points close to the coordinate origin but in general, the values do not depend on the actual software changes (RWC). Rather they are widely spread throughout the plot. Even for small RWC values large ADN values are obtained. For very few large change values small ADN values are computed.

- For T_{St}, T_{Sq}, T_{P_M}, T_{P_S} and $T_{P_{SS}}$ the plots show higher frequency of large ADN values and higher magnitudes of these large values compared to the EVOSTREETS variants and T_{SD}. Frequency and magnitudes differ between the approaches. While for $T_{P_{SS}}$ many values with high magnitude are shown, for T_{St} only few values with modestly increased ADN values are shown.

The DISTRIBUTION ANALYSIS showed that ADN values for E_L, E_G, E_P, and T_{SD} in general are rather low. However, values above 0.1 are possible for these approaches as well. The plots depicted here do not show such values above 0.1. Therefore the data suggest that large layout adaptations for E_L, E_G, E_P, and T_{SD} only occur if large software changes are applied, i.e. small changes do not cause large layout adaptations.

The plot for P clearly shows two characteristics: First, small changes to the underlying data can cause far node displacements. Second, small changes to the underlying data rather often cause far node displacements.

CONCLUSIONS FOR ADN

Above, three analyses were executed to determine the average displacement of nodes (ADN). From these analyses, the following conclusions can be drawn:

- On average, nodes move least for E_L, E_G, and T_{SD}. The Top Performer analysis showed that the lowest ADN values for most layout transitions are achieved by one of these approaches. The distribution analysis showed that the ADN values for these approaches typically are rather low. Also, they are not susceptible to small changes.
- E_P slightly falls behind the other EVOSTREETS approaches with respect to the Top Performance analysis, but concerning the distribution of ADN values, these differences are rather small. Also it offers a similarly good behavior concerning small changes.
- P clearly offers the worst performance of all approaches. It has the lowest top performance and hardly reaches a top three ranking. Its ADN values are usually much higher, and it suffers from strong susceptibility to small changes.

Nearest Neighbor Within (NNW)

TOP PERFORMER

Tables 5.5 and 5.6 show the results of the Top-Performer analysis for the NNW measure. The following observation can be made:

- E_G performs best for 303 layout transitions (table 5.5), followed by T_{SD} (112 transitions), and E_L (95 transitions). E_P clearly falls behind E_G, E_L, and T_{SD}. However, it still performs better than the P and the remaining treemap approaches.
- Concerning the Top-Three-Performance in table 5.6, E_G and E_L perform equally well, followed by T_{SD}. E_P again falls behind but still performs better than the remaining treemap approaches and P.
- P, T_{P_M}, T_{P_S}, $T_{P_{SS}}$, T_{Sq}, and T_{St} only occasionally reach a top three ranking.
- While the distribution of best performances among layouts differs for each system, it is always E_G, T_{SD}, or E_L that perform best regarding the number of best performances for a particular system (Top-1 and Top-3).

E_G clearly performs best. In the majority of cases, it achieves the lowest NNW value, which is about 59% of all layout transitions. Neither T_{SD} nor E_L (which performed better with respect to ADN) can provide a similar performance. E_P performs best in only 16 cases; for only 100 cases, it reaches a top three ranking.

System	E_L	E_G	E_P	T_{SD}	T_{St}	T_{Sq}	T_{P_M}	T_{P_S}	$T_{P_{SS}}$	P
Apache Ant	1	15	0	5	0	0	0	0	0	0
Apache CXF	14	23	0	5	0	0	0	0	1	0
ArgoUML	1	9	0	5	0	0	0	0	0	0
Compass	7	10	1	1	0	0	0	0	0	0
Datanucleus Core	9	27	2	9	0	0	2	2	0	0
Hibernate Core	2	20	1	16	0	0	0	0	2	0
JFreeChart	1	37	2	14	0	0	0	0	0	0
Mule Core	8	26	2	9	0	0	1	0	0	0
Neo4J	7	3	1	15	0	1	1	1	1	0
Spring Framework	1	45	2	4	0	0	0	0	0	0
JME 3	3	5	0	11	0	0	0	0	0	0
JMol	9	5	0	0	4	1	5	2	0	0
Checkstyle	3	17	0	5	0	0	0	0	0	0
CrocoCosmo	25	14	3	6	0	0	2	0	0	0
FindBugs	0	16	0	5	0	0	0	0	0	0
ProcessDash	4	31	2	2	0	0	1	0	0	0
$\sum$	95	303	16	112	4	2	12	5	4	0

Table 5.5: Top Rankings for NNW

System	E_L	E_G	E_P	T_{SD}	T_{St}	T_{Sq}	T_{P_M}	T_{P_S}	$T_{P_{SS}}$	P
Apache Ant	19	20	0	19	0	0	3	1	0	0
Apache CXF	41	40	1	40	0	0	0	0	1	0
ArgoUML	15	15	4	11	0	0	0	0	0	0
Compass	18	18	6	9	0	0	3	0	0	0
Datanucleus Core	43	43	8	29	3	1	7	6	1	0
Hibernate Core	36	40	4	34	1	0	4	2	2	2
JFreeChart	50	50	3	44	0	0	3	0	0	0
Mule Core	42	42	4	38	1	0	4	0	1	0
Neo4J	24	22	1	22	2	1	1	2	1	0
Spring Framework	48	48	14	36	0	0	0	0	0	0
JME 3	18	18	1	17	0	0	0	0	0	0
JMol	14	15	3	5	8	5	11	6	2	0
Checkstyle	25	25	13	12	0	0	0	0	0	0
CrocoCosmo	46	41	24	21	3	2	6	1	3	0
FindBugs	21	21	1	20	0	0	0	0	0	0
ProcessDash	35	36	13	25	1	0	1	0	0	0
$\sum$	495	494	100	382	19	9	43	18	11	2

Table 5.6: Top 3 Rankings for NNW

In comparison to ADN, E_P even more clearly falls behind the other EVOSTREETS variants and T_{SD}.

For P and all treemaps except T_{SD}, the number of times they perform best or reach at least a top three ranking is very low compared to the other approaches. Thus, treemaps perform worse than all the other approaches with respect to NNW. Similarly to ADN, P performs worst of all approaches for NNW, too. For no transition it reaches the lowest NNW value, and it is placed among top three performers twice only.

DISTRIBUTION ANALYSIS
A complete set of diagrams is provided in the appendix B, pp. 206(ff). Similarly to ADN, the magnitudes covered by the distribution boxplots vary strongly among systems, but despite this difference some patterns are apparent for all systems. The following observations can be made:

- E_L, E_G, and T_{SD} have lowest median and average values and an interquartile range that covers lower NNW values than the other approaches. For one system only (JMol), T_{P_M}, T_{P_S}, T_{Sq}, and T_{St} offer a similar characteristic.
- E_P performs worse than the other EVOSTREETS approaches. Both, median and average values are larger than those of E_L and E_G. Also, for all systems its interquartile range covers larger values.
- There is no clear difference between P and the remaining treemaps.

The plots confirm the result from the TOP PERFORMANCE analysis that E_P performs worse than the other EVOSTREETS variants. On the other hand, they lessen the top performance observation for E_G since the actual values for E_G, E_L, and T_{SD} cover rather similar values, especially compared to the other approaches.

Compared to the ADN discussion, P closes the gap to the treemaps. Although P suffers from frequent large node displacements, it preserves node neighborships to a similar extent as the pivot based treemaps, T_{Sq}, and T_{St}.

SUSCEPTIBILITY TO SMALL CHANGES
Figures 5.3 and 5.4 show the plots for P and E_L. A complete set of diagrams for all approaches is provided in appendix B. The following observations can be made:

- E_L, E_G, and T_{SD} show a similar behaviour. The NNW values increase for larger change values. For E_L, a few small changes cause NNW values around 0.2, whereas for T_{SD} some values above 0.2 are detected for larger change values.
- In contrast to E_L and E_G, for E_P NNW values are above 0.1 even for small change values. Also, for E_P high NNW values of up to 0.5 can be obtained for very small software changes.
- For P, T_{St}, T_{Sq}, T_{P_M}, T_{P_S}, and $T_{P_{SS}}$, magnitudes and frequency of high values is higher than for E_L, E_G, T_{SD}, and E_P. For T_{St} we additionally observe a slightly lesser increase.

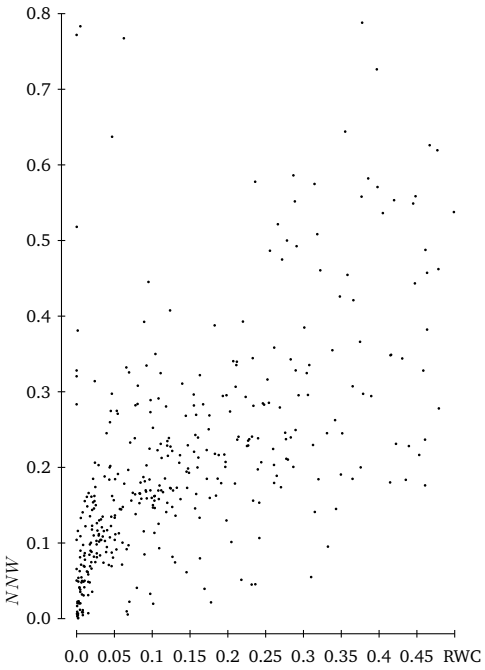


Figure 5.3: NNW Plot for P

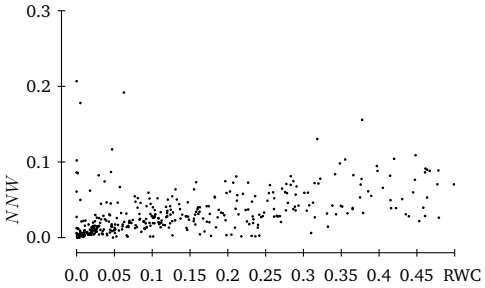


Figure 5.4: NNW Plot for E_L

In general, these plots show that larger changes by trend induce larger layout adaptations. However, there are differences between the approaches: For E_L, E_G, and T_{SD} small changes typically do not induce large layout adaptations. While these approaches in general do not yield high NNW values, the Distribution Analysis above showed that values above 0.3 are possible. These relatively large layout adaptations do however only occur for reasonably large software changes.

CONCLUSIONS FOR NNW

Concerning the Nearest Neighbor Within (NNW) measure, the following conclusions can be drawn from the above analyses:

- Of all approaches, E_L, E_G, and T_{SD} best preserve neighborships. A slight trend towards E_G is visible as the Top Performer analysis showed.
- E_P performs worse than E_L, E_G, and T_{SD}. Usually, it causes more changes of neighborships and is less resistant to small software changes.
- Compared to the other approaches, P, T_{St}, T_{Sq}, T_{P_M}, T_{P_S}, $T_{P_{SS}}$ do not preserve neighborships very well. As the analyses show, these approaches typically cause relatively large layout adaptations and are more susceptible to small changes.

Ranking

TOP PERFORMER

Tables 5.7 and 5.8 show the results of the Top-Performer analysis for the $Ranking$ measure. The following observation can be made:

- E_G performs best for 201 layout transitions (table 5.7), followed by E_L (179 transitions), and T_{SD} (120 transitions). E_P again clearly falls behind E_G, E_L, and T_{SD}. It performs better than P and the remaining treemap approaches.
- Results are similar concerning the top three performances shown in table 5.8, E_G and E_L perform similarly well, followed by T_{SD}. E_P again falls behind but still performs better than the remaining treemap approaches and P.
- P, T_{P_M}, T_{P_S}, $T_{P_{SS}}$, T_{Sq}, and T_{St} only occasionally reach a top three ranking.

E_G performs slightly better than E_L with respect to the number of best performances. However, the difference between both approaches is less than for NNW. Also, both reach a top three ranking for a comparably number of cases. Thus, concerning $Ranking$, E_G and E_L can be considered to perform equally well. Besides that, the results for $Ranking$ are similar to the results for NNW: T_{SD} shows a slightly lower performance compared to E_G and E_L, but still preserves the order better than E_P which performs best in 16 cases only and reaches a top three ranking in 167 cases, only. Thus, E_P again performs worse than E_G, E_L, and T_{SD}.

Concerning P and all treemaps except T_{SD}, similar conclusions to NNW can be drawn: All these approaches seldom perform best or reach at least a top three ranking. Thus, P, T_{P_M}, T_{P_S}, $T_{P_{SS}}$, T_{Sq}, and T_{St} clearly perform worse with respect to NNW than the EVOSTREETS approaches and the Slice&Dice treemap.

System	E_L	E_G	E_P	T_{SD}	T_{St}	T_{Sq}	T_{P_M}	T_{P_S}	$T_{P_{SS}}$	P
Apache Ant	5	8	0	7	0	0	0	0	0	0
Apache CXF	14	22	0	5	0	0	0	0	0	0
ArgoUML	2	6	0	7	0	0	0	0	0	0
Compass	9	2	0	6	0	0	1	0	0	0
Datanucleus Core	13	19	2	12	0	0	0	0	0	0
Hibernate Core	9	15	0	17	0	0	0	0	0	0
JFreeChart	16	21	1	15	0	0	0	0	0	1
Mule Core	13	25	0	5	0	0	0	0	0	0
Neo4J	6	9	0	10	0	0	0	0	0	0
Spring Framework	33	8	3	4	0	0	0	0	0	0
JME 3	7	5	2	4	0	0	0	0	0	0
JMol	3	7	1	7	1	0	0	2	0	0
Checkstyle	10	8	0	7	0	0	0	0	0	0
CrocoCosmo	13	25	3	7	0	0	0	0	0	0
FindBugs	11	9	0	1	0	0	0	0	0	0
ProcessDash	15	12	4	6	0	0	0	0	0	0
$\sum$	179	201	16	120	1	0	1	2	0	1

Table 5.7: Top Rankings for $Ranking$

System	E_L	E_G	E_P	T_{SD}	T_{St}	T_{Sq}	T_{P_M}	T_{P_S}	$T_{P_{SS}}$	P
Apache Ant	20	20	2	18	0	0	0	0	0	0
Apache CXF	37	38	11	30	5	0	0	1	1	0
ArgoUML	10	14	4	14	1	1	0	1	0	0
Compass	17	14	7	11	1	0	4	0	0	0
Datanucleus Core	38	38	8	34	7	1	6	5	1	0
Hibernate Core	36	37	4	33	3	1	4	1	3	1
JFreeChart	48	45	9	43	2	0	3	1	0	1
Mule Core	39	41	20	21	3	0	1	0	0	1
Neo4J	19	23	1	23	6	2	0	1	1	0
Spring Framework	47	43	21	28	2	0	1	2	0	0
JME 3	14	17	8	12	3	0	0	0	0	0
JMol	10	19	3	17	7	1	2	4	0	0
Checkstyle	25	25	12	13	0	0	0	0	0	0
CrocoCosmo	38	40	32	21	5	1	5	1	1	0
FindBugs	21	21	3	18	0	0	0	0	0	0
ProcessDash	34	33	22	16	0	0	1	2	0	0
$\sum$	453	468	167	352	45	7	27	19	7	3

Table 5.8: Top 3 Rankings for $Ranking$

DISTRIBUTION ANALYSIS

A complete set of diagrams is provided in the appendix B, pp. 213(ff). All diagrams are logarithmically scaled. Similarly to ADN and NNW, some patterns are apparent for all systems, although the magnitudes covered by the distribution boxplots vary strongly among systems. The following observations can be made:

- E_L, E_G, E_P, and T_{SD} have lower median values, lower average values, and interquartile ranges covering lower values than the other approaches. For one system only (JMol), T_{P_M}, T_{P_S}, T_{Sq}, and T_{St} offer similar characteristics concerning the interquartile range and median values. For T_{P_M} and T_{P_S} the average values are slightly increased due to some extreme values. For one more system (Neo4J), E_P and the treemap approaches T_{P_M}, T_{P_S}, T_{Sq}, and T_{St} show a similar performance.
- For all systems, the median and average values for E_P are higher than those of E_L and E_G. Additionally, its interquartile ranges usually cover larger values than the corresponding interquartile ranges for E_G and E_L.
- For every system, P has the highest median and the highest average value. For all systems, its interquartile range covers larger values than the other approaches.

The distribution analysis clearly shows the inability of the P approach to preserve the order of nodes. Its $Ranking$ values are typically very high in comparison to the other approaches. Also the analysis shows that the EVOSTREETS approaches and the Slice&Dice treemap outperform all other approaches. Among the EVOSTREETS approaches, we see again that E_P falls behind E_G and E_L. The diagrams do not show an apparent difference between E_G and E_L which means that this simple grid based packing usually does not disturb the ordering of nodes.

SUSCEPTIBILITY TO SMALL CHANGES

Figures 5.5 and 5.6 show the plots for P and E_L. A complete set of diagrams for all approaches is provided in appendix B. The following observations can be made:

- The plots for E_L, E_G, and T_{SD} show similar point clouds. Values are typically smaller than 10, only few points exceed this border. Particularly for very small change values a value higher than 10 is seldom detected. Values increase for larger change values. Additionally, the plots show point clusters close to the coordinate origin.
- Values for P are slightly higher than for E_L, E_G, and T_{SD}. Also, the plot does not show a clear point cluster close to the origin.
- For P the plot shows a large empty region at the bottom right. Only for small RWC values, small $Ranking$ values are depicted. Points are rather uniformly spread throughout the top of the plot for all RWC values. The plot does not depict any accumulations of points.
- For T_{St}, T_{Sq}, T_{P_M}, T_{P_S}, and $T_{P_{SS}}$, magnitudes and frequency of large values are much higher than for the EVOSTREETS approaches and the Slice&Dice treemap.

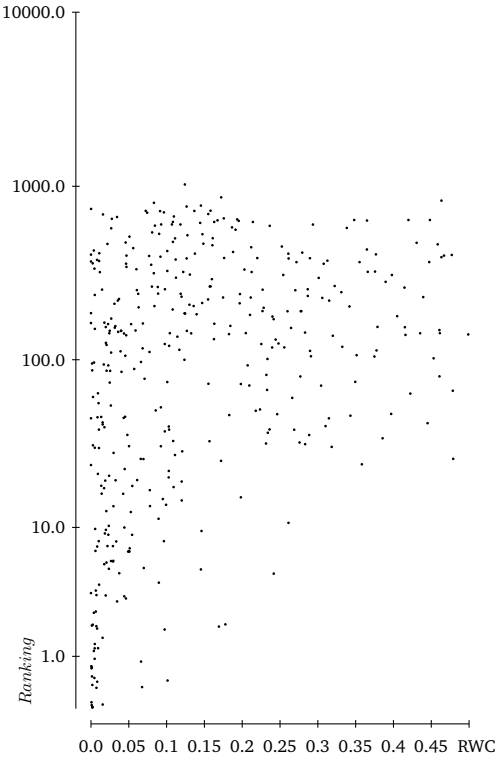


Figure 5.5: $Ranking$ Plot for P

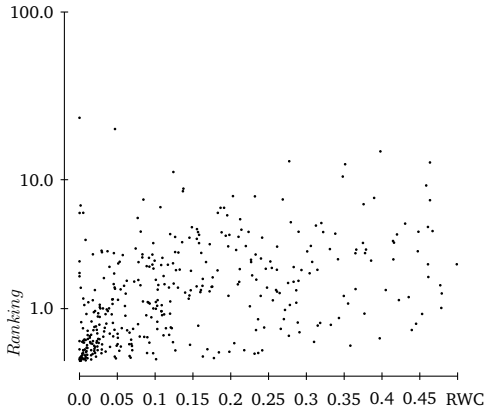


Figure 5.6: $Ranking$ Plot for E_L

The concentration of points close to the coordinate origin for E_L, E_G, and T_{SD} shows that these approaches are less susceptible to small changes of the underlying data than those approaches for which such a concentration is not observable. Small changes usually do not imply large node perturbations.

In contrast, for P no concentration of points close to the coordinate origin can be detected. Rather, the large empty region at the bottom right indicates that small layout adaptations can only occur for small software changes. On the other hand, small software changes do not imply small layout adaptations. Instead, even very small changes to the underlying data can induce the very large node perturbations.

CONCLUSIONS FOR RANKING
Concerning the Ranking measure, the following conclusions can be drawn from the above analyses:

- The analyses showed that E_L, E_G, and T_{SD} best preserve the order of nodes. In nearly all cases, one of those approaches performs best. Also, changes of order are typically rather low for these approaches, and small changes do not cause large layout adaptations.
- E_P again falls behind E_L, E_G, and T_{SD}, but it preservers order better than P, T_{St}, T_{Sq}, T_{P_M}, T_{P_S}, and $T_{P_{SS}}$.
- P worst preserves the order of nodes. It suffers from frequent large node perturbations which can be triggered even by very small changes to the underlying data set.

5.2.5 Summary and Conclusions

Layout consistency is an important quality criterion in many application scenarios handling evolving data sets. Inconsistent layouts increase the error-proneness and effort for data monitoring and interpretation. Different approaches for visualizing hierarchically structured data are available. Above, these approaches have been investigated with respect to the layout consistency they offer. Layout consistency is a multi-facetted property, i.e. layout inconsistencies arise from different reasons such as large node displacements, or changes to node neighborships and node orderings. For several layout approaches, these properties were characterized by analyzing the top performance, typical behavior, and susceptibility to small changes. From these analyses, the following conclusions regarding the layout consistency are drawn:

- The analyses of ADN, NNW, and $Ranking$ clearly show that nodes move least, and neighborship and node ordering are preserved best for E_L, E_G, and T_{SD}. From these observations we conclude that E_L, E_G, and T_{SD} offer higher layout consistency for evolving data sets.
- Rectangle packing that targets towards high compactness clearly falls behind all other approaches: Nodes typically move large distances if the represented data changes. Also, node neighborships and node orderings are strongly disturbed. All these effects occur for even very small changes to the represented data.

- The analyses also show that the effects of combining the EVOSTREETS approach with rectangle packing algorithms on layout consistency strongly depend on the packing algorithm used. Whereas the use of a simple grid based algorithm does not reduce layout consistency, combining the EVOSTREETS with the P approach (which optimizes layout compactness) clearly increases average node displacements and disturbs node neighborships and orderings.

5.3 Layout Effectiveness

As discussed in section 4.3.2, previous research has identified several layout characteristics that may increase or decrease the effort and error-proneness of information reading: Low readability of hierarchical data ([13], [184]) can be avoided using offsets between node representations. Thus it is not investigated further. Concerning layout effectiveness, preserving the input order ([23]) and continuity ([179], [188]) are less relevant layout characteristics in the context of this work because the input order usually has no meaning that could be represented in the layout. Thus, both characteristics are not investigated further. Also, we do not further analyze the layouts using the *Readability* measure by [23] for two reasons: First, it is not apparent if and to which degree a frequent eye motions during sequential scanning of layout rectangles improves the effort or reduces error-proneness of cognizing the structure of a treemap in general. Second, the metric is based on threshold values which are not confirmed by empirical results.

Two characteristics are analyzed in this section: layout compactness and aspect ratios.

5.3.1 Compactness

Compactness is an important effectiveness criterion as it supports in overviewing large data sets quickly. The goal of this section is to determine typical compactness characteristics for E_L, E_G, E_P, and P. Treemaps are not considered because they are space-filling.

Measures

Compactness can be measured directly by determining the fraction of the entire layout space which is actually used for representing data. The actually used space can easily be computed as the aggregate area of all node representations in the layout. The “entire layout space”, however, is difficult to outline as there is no apparent natural border enclosing the layout. Thus, this border must be defined on the basis of some enclosing shape like the smallest rectangle, the smallest convex polygon, or the smallest circle which enclose all points of the layout. All these shapes are convex which means that for any two points inside the shape area their direct connection is fully surrounded by the shape as well. Figure 5.8 illustrates the differences between these shapes.

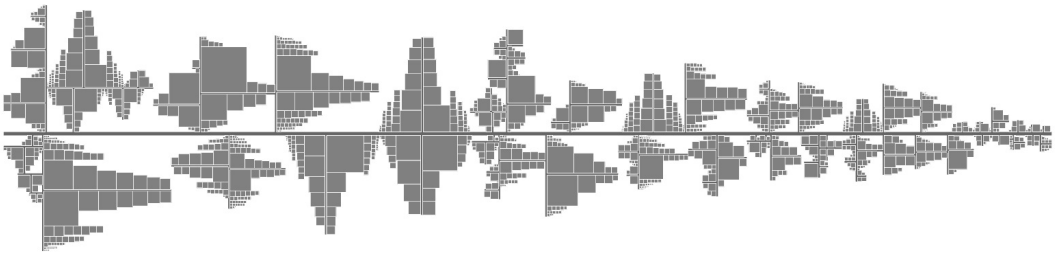
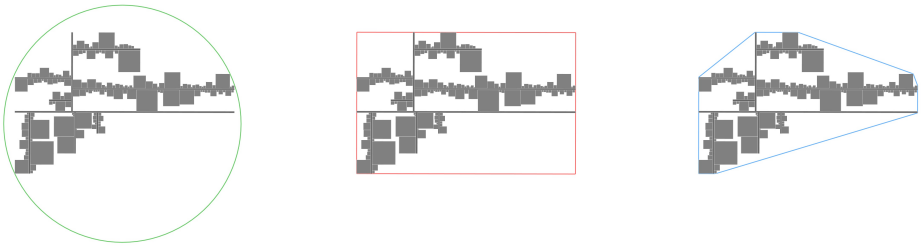


Figure 5.7: E_G layout for Hibernate Core



(a) Circle (b) Rectangle (c) Polygon

Figure 5.8: Enclosing Shapes

Given these shapes, corresponding compactness measures can easily be defined as the fraction of the surrounding shape that is occupied by city elements like buildings and streets. We use A_N to refer to the actually occupied area, whereas A_C, A_R, and A_P denote the areas of the surrounding circle, rectangle, and polygon, respectively. On the basis of the smallest enclosing circle, we define a compactness metric $SECC$ (*Smallest Enclosing Circle Compactness*) as $SECC := 100 * \frac{A_N}{A_C}$. In an analogous manner, compactness metrics on the basis of the smallest enclosing rectangle and polygon can be defined. We use the terms $SERC$ and $SEPC$ for the corresponding rectangle and polygon based compactness measures.

The values of these measures can differ strongly. For the layout depicted in figure 5.7, we obtain 8.9% for $SECC$, 30.5% for $SERC$, and 45.3% for $SEPC$. Thus, depending on which shape is used, very different characterizations concerning layout compactness could be obtained. Intuitively, the layout in figure 5.7 appears to be rather compact because only few empty space between node representations occurs. Clearly, $SECC$ does not support this characterization. On the other hand, the layout is unbalanced in the sense that its width is much larger than its height. For some applications as depiction and/or interactive rotation on screen, such longish layouts require a larger zoom than similarly compact but better shaped layouts to be fully displayable on an output device. Clearly, $SEPC$ does not support this characterization.

While $SEPC$ approximates a best compactness characterization, $SECC$ better accounts for imbalance. Thus, to obtain a comprehensive compactness characteriza-

tion of the above layout approaches, we analyze them using all of the above measures.

Analysis

For all systems listed in table 5.2, all EVOSTREETS layouts (E_L, E_G, E_P) and the rectangle packing layout P are computed. Treemaps are not considered because they are space-filling and thus in all cases yield maximum $SERC$, $SEPC$, and $SECC$ values. For $SECC$, a square treemap always reaches a value of $2/\pi$ which is about 63%. Layouts are computed not only for one particular version (e.g. the first or the last), but for all versions of the respective system.

The $SECC$, $SERC$ and $SEPC$ values of all layouts are computed. Applying all measures to 531 versions visualized using four layout approaches yields 6372 data points. We analyze $SEPC$ first and compare the results with $SECC$ and $SERC$ afterwards. The respective data points are grouped by system, and analyzed using a box plot representation that characterizes the distribution of compactness values. To get a qualitative understanding of the compactness values, example layouts are provided as well.

For reasons of readability, district offsets and street widths can be adjusted by users to satisfy individual preferences. To avoid an influence of these individual settings on layout consistency the following settings are used for the analysis:

- For all EVOSTREETS approaches, the width of inner tree nodes' line representations (streets) is set to zero. No offsets between rectangles are used for EVOSTREETS and teh rectangle packing approach P.
- For all EVOSTREETS approaches, elevation is not used, the elevation level increment is set to zero. Elevation levels may in fact reduce the compactness of the overall layout, but since they are an optional feature of the EVOSTREETS approach, they are not regarded during the analysis of layout compactness.
- Representations for removed nodes are counted as used space.

Results and Discussion

$SEPC$

Figure 5.9 shows the results for $SEPC$ by system. Due to space limitations the box plots are not labeled, but the same order of layouts (E_L, E_G, E_P, P from left to the right) is used for all diagrams. The magnitudes covered by the distribution boxplots only slightly vary among systems. Some patterns can be detected:

- All characteristics except the maximum value (minimum value, average and median value, interquartile range) are higher for P than for all EVOSTREETS approaches. For Checkstyle, FindBugs, and ProcessDash, the maximum $SEPC$ values are higher for E_P (Checkstyle, FindBugs) and E_L, E_G (ProcessDash), respectively.

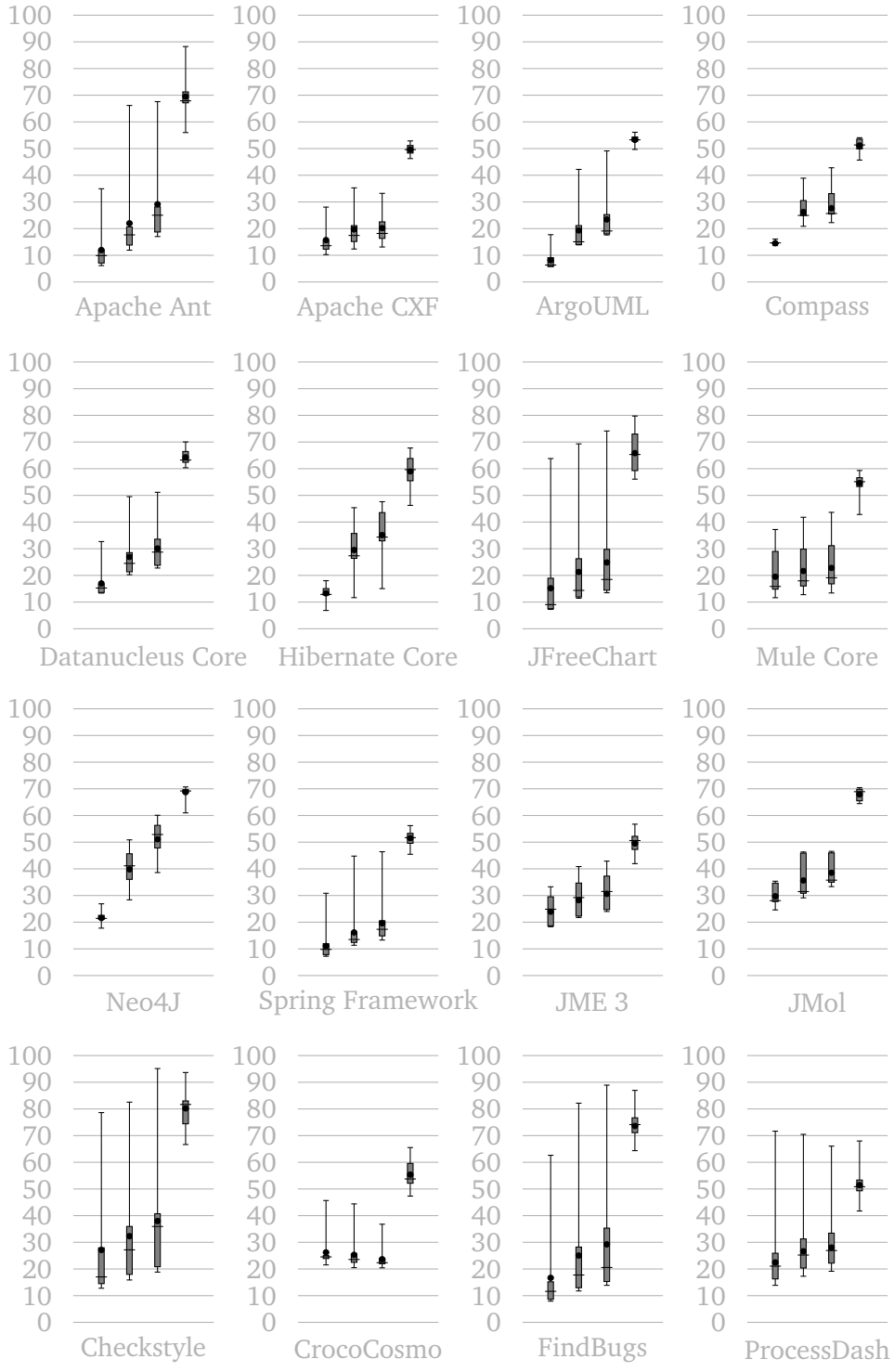


Figure 5.9: SEP Compactness for E_L, E_G, E_P, P

- Besides very few exceptional cases, $SEPC$ values (minimum, maximum, average, median, and interquartile range) for E_P are higher than E_G, and $SEPC$ values for E_G are higher than E_L. The differences between E_L, E_G, E_P vary among systems. Whereas they are rather larger for Neo4J, for Apache CXF and Mule Core these differences are rather low.

- For CrocoCosmos, E_G and E_P yield lower $SEPC$ values as compared to E_L with respect to minimum, maximum, average, and median values, and the values covered by the interquartile range.

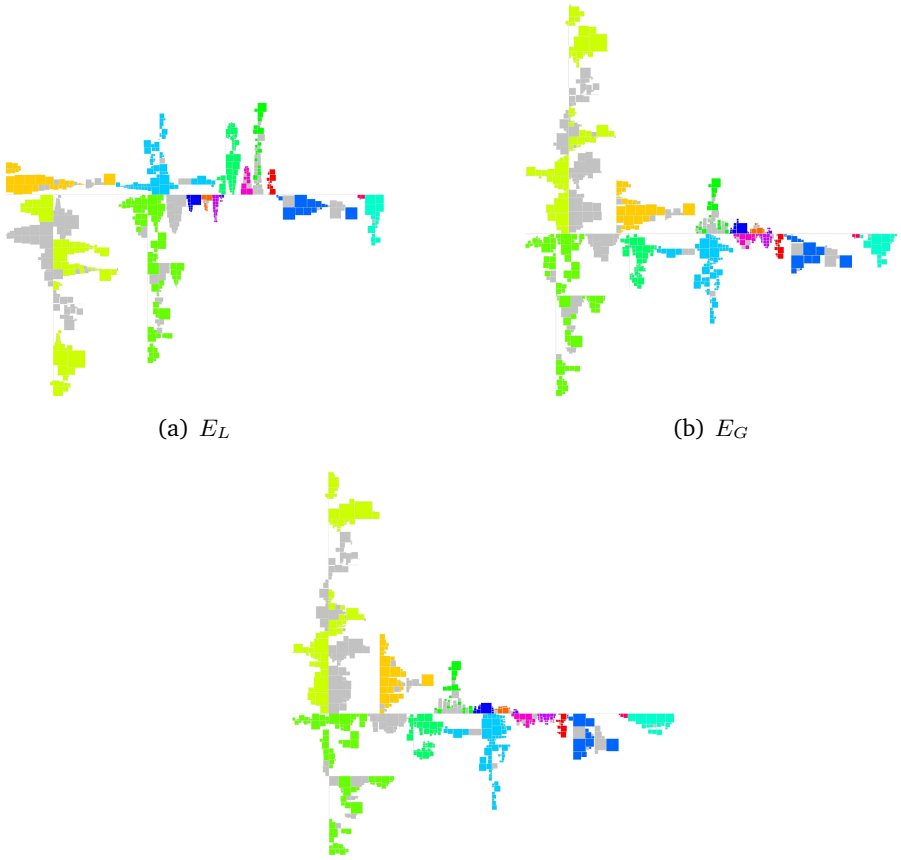
- For some systems (Ant, JFreeChart, Checkstyle, FindBugs, ProcessDash), the EVOSTREETS approaches yield considerably large maximum values while on the other hand the interquartile range for these systems still covers rather small values.

As the diagrams clearly show, P outperforms all EVOSTREETS approaches. For all systems, it performs better and its interquartile range rarely covers values below a 50% $SEPC$. Also, its interquartile range is rather narrow, and the median and average values indicate that these magnitudes are typical for P.

Concerning the EVOSTREETS approaches, a clear ordering can be derived. E_L typically shows a lower compactness than E_G which in turn shows a lower compactness than E_P. Hence, the use of a rectangle packing algorithm in the EVOSTREETS approach usually improves layout compactness. While these values in general are rather small (particularly in comparison to the magnitudes for P), the improvements obtained by using an additional rectangle packing often indicate a doubling of $SEPC$ values.

For CrocoCosmos we see that E_L yields a higher compactness than E_G and E_P. The reasons for this can be seen in figure 5.10 which depicts the E_L, E_G, and E_P layouts for the last version of CrocoCosmos. We observe that for E_L the two longest top-level streets (depicted in light green and dark green, respectively) are placed next to each other on the same side of the main street. In contrast, for E_G and E_P these streets are placed on distinct sides of the main street which (due to their length) results in an increased layout height compared to E_L. This layout difference results from the simple heuristic that is applied to assign elements to both sides of the containing nodes line representation: When laying out a graph node, all its children are laid out first. Afterwards, they are sorted by width (not length!) and assigned to both sides of the node's line representation. Clearly, effects like these can be avoided in advance by using more sophisticated layout heuristics. However, the EVOSTREETS variants used in this thesis use the simple heuristic which assigns child representations on the basis of their width only.

For Ant, JFreeChart, Checkstyle, FindBugs, and ProcessDash, diagram 5.9 shows noticeably large maximum values for the EVOSTREETS approaches. A manual inspection showed that these large $SEPC$ values are obtained only for the first system versions. Since in these versions the respective systems consist of only very few classes (see table 5.2), the corresponding layouts are rather small. There are no



(a) E_L (b) E_G

(c) E_P

Figure 5.10: E_L, E_G, E_P Layouts for CrocoCosmos, Version: 51

large and unbalanced tree structures which need to be laid out. Consequently, these large compactness values are exceptional cases.

$SECC$ and $SERC$
Similar analyses can be done for the $SECC$ and $SERC$ compactness measures. The corresponding diagrams B.18 and B.17 are given in appendix B. Each of the diagrams covers 2124 layouts (531 versions visualized with four approaches).

We obtain similar results for $SECC$ and $SERC$, although the absolute values for $SECC$ are smaller than for $SERC$ and $SEPC$: The characteristics of the boxplots (median, minimum value, maximum value, and average value) again indicate that layouts based on the rectangle packing algorithm P are clearly more compact than EVOSTREETS layouts. E_G and E_P perform better than E_L. For CrocoCosmos again we see the same inverse situation already discussed above, and for initially small systems relatively large compactness values can be observed.

Example Layouts
The above discussion revealed that the EVOSTREETS approaches E_L, E_G, and E_P

suffer from a lower compactness than the rectangle packing approach P. Also, the discussion revealed that the additional use of a rectangle packing algorithm in the EVOSTREETS approaches (E_G and E_P) usually increases the layout compactness. But we do not yet understand what these values actually mean. To get a qualitative understanding of layout compactness, we assess those layouts for which the lowest $SEPC$ compactness values were determined.

Figures 5.11, 5.12, and 5.13 each show the E_L and E_G layouts for the systems Ant, Spring Framework, and ArgoUML. As the figures illustrate, all E_L layouts suffer from a rather low compactness that is caused by long streets. Also the figures show, that the use of a grid based rectangle packing clearly avoids these long streets and thus increases the compactness of the layout significantly.

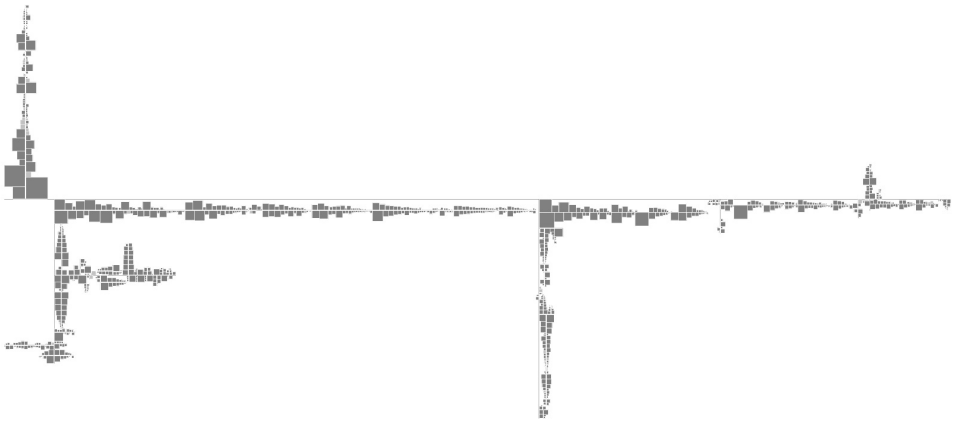
Clearly, an assessment of the usability of EVOSTREETS based visualizations finally must be left to the users who actually use them. But from our perspective, the E_L layouts depicted in figures 5.11, 5.12, and 5.13 show serious deficiencies with respect to layout compactness which are mainly caused by long streets. E_G layouts, however, clearly increase the layout compactness because they avoid these long streets. The layouts depicted in figures 5.11, 5.12, and 5.13 are reasonably compact even though they are those layouts for which we observed the lowest $SEPC$ compactness values. Thus, from our perspective, E_G layouts seem to provide a sufficiently high compactness for the example systems listed in table 5.2.

The positive influence of a rectangle packing in the EVOSTREETS approach (e.g. E_G) is diminished again by the evolutionary segmentation, i.e. the overall layout compactness gets worse again as soon as the temporal resolution of the underlying software model is increased.

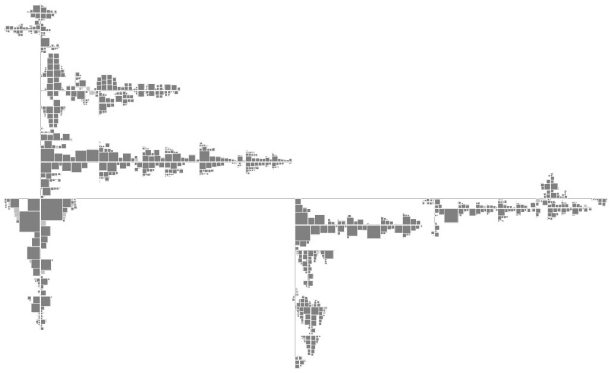
5.3.2 Aspect Ratios

In general, treemaps are space filling visualizations and thus outperform both the EVOSTREETS and rectangle packing approaches described above. On the other hand, they suffer from problems of readability of hierarchical data which can however easily be resolved using additional offsets. The above analyses showed that particularly Slice&Dice treemaps usually outperform all other treemaps with respect to layout consistency and even offer a similar quality as the EVOSTREETS approaches E_L and E_G. Therefore, Slice&Dice appears to be reasonable candidate for building consistent software cities.

Previous research (e.g. [23], [179]), however, showed that especially Slice&Dice treemaps tend towards high aspect ratios which negatively impact on layout effectiveness as they cause low visibility of rectangles in the layout ([23]). More precisely, previous research actually showed that Slice&Dice treemaps tend towards high *average* aspect ratios. Conclusions regarding the typical magnitudes must be drawn carefully because high average aspect ratios may be caused by a few extreme values. To characterize typical aspect ratio magnitudes for a treemap approach more comprehensively the distribution of aspect ratios must be considered.

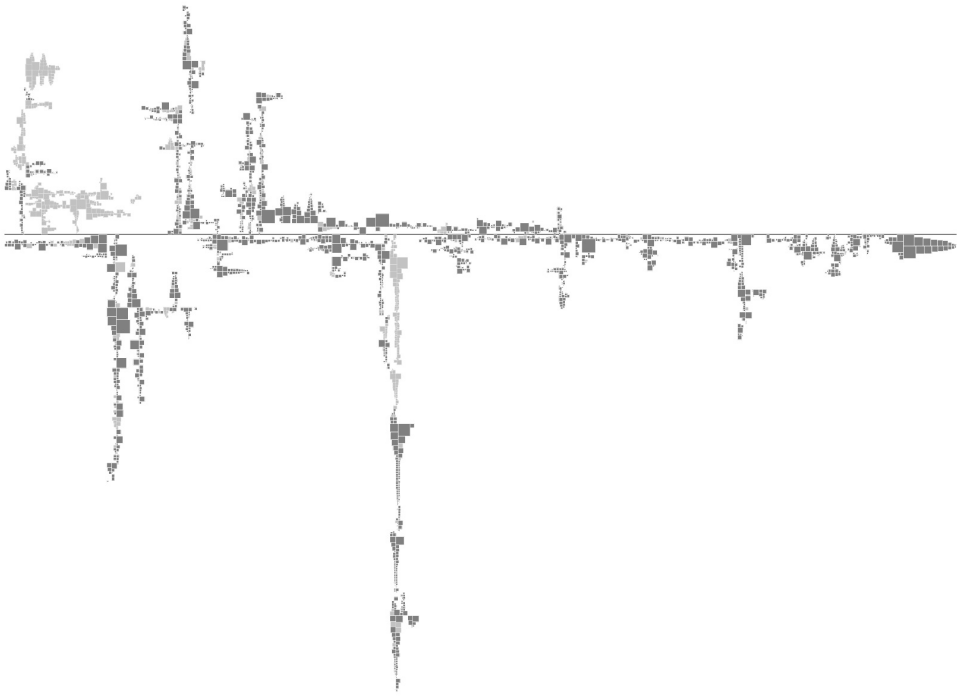


(a) E_L: $SEPC = 6.29$, $SERC = 4.01$, and $SECC = 2.12$

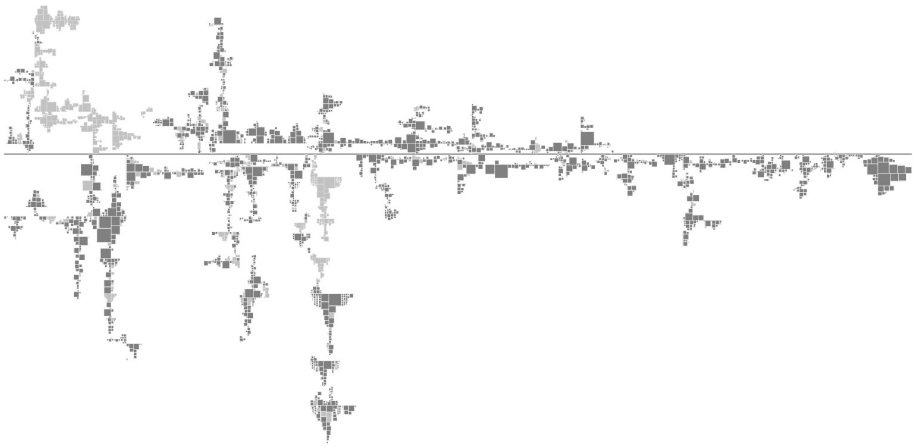


(b) E_G: $SEPC = 11.91$, $SERC = 7.22$, and $SECC = 5.03$

Figure 5.11: E_L and E_G Compactness for Ant (21 versions, 1044 JAVA classes)

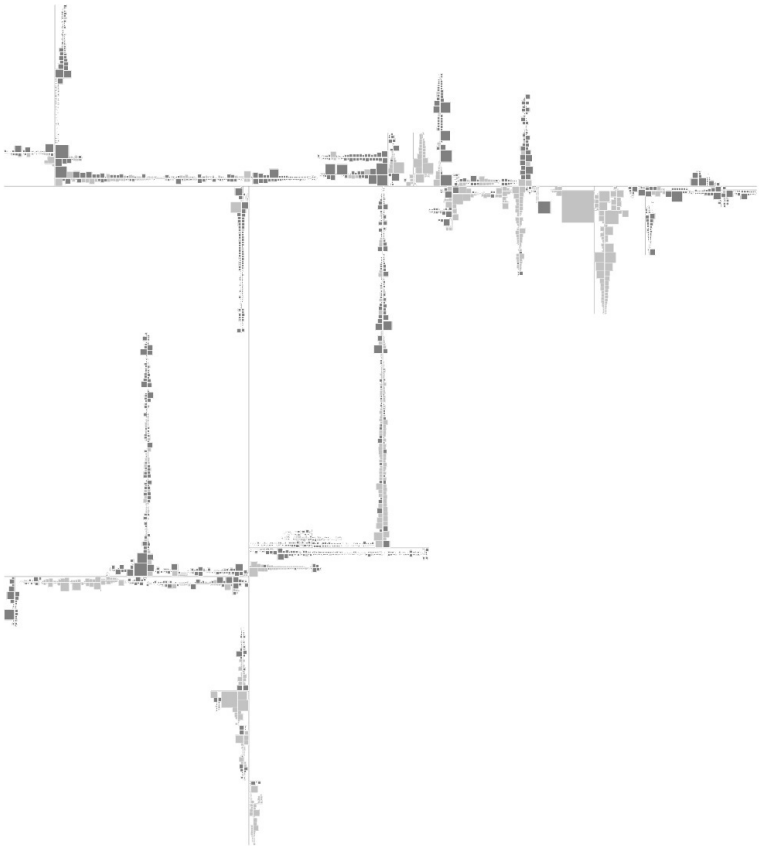


(a) E_L: $SEPC = 7.23$, $SERC = 4.06$, and $SECC = 3.67$

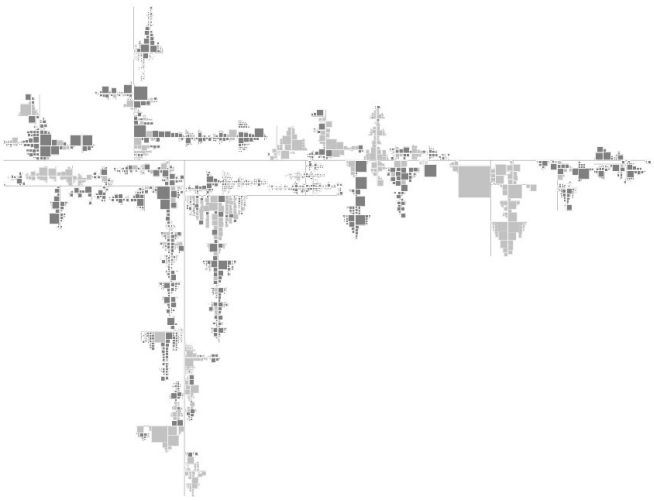


(b) E_G: $SEPC = 11.35$, $SERC = 6.69$, and $SECC = 4.05$

Figure 5.12: E_L and E_G Compactness for Spring (49 versions, 2150 JAVA classes)



(a) E_L: $SEPC = 5.77$, $SERC = 3.45$, and $SECC = 3.21$



(b) E_G: $SEPC = 13.91$, $SERC = 7.42$, and $SECC = 7.01$

Figure 5.13: E_L and E_G Compactness for ArgoUML (16 versions, 1921 JAVA classes)

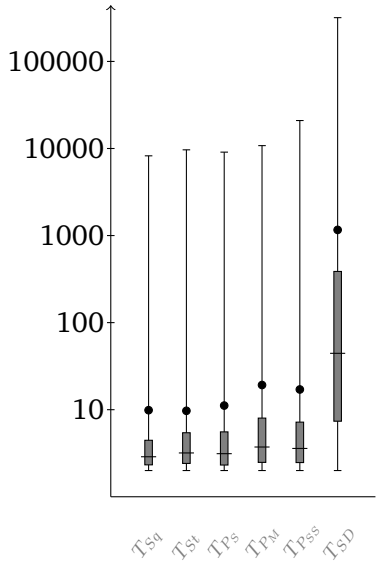


Figure 5.14: Box-Plots of Aspect Ratios by Treemap Approach

Besides aspect ratio magnitudes, another related characteristic (which is rarely discussed in the literature) impacts on layout effectiveness: *Comparing* treemap rectangles by size is difficult not only because of high aspect ratios (which actually diminish rectangle *visibility*), but also because of a large variance of aspect ratios in one and the same layout. In general, layout rectangles are difficult to compare in size if their aspect ratios deviate from each other. If on the other hand, the aspect ratios of all layout rectangles of a given treemap layout were the same, then a comparison of node sizes reduces to a comparison of node length or width, respectively, i.e. the comparison reduces to one dimension.

In summary, there are two problems arising from variable aspect ratios: low visibility and low comparability. Whereas low visibility of layout rectangles is caused by high aspect ratios, low comparability of layout rectangles is caused by a large variance of aspect ratios. The goal of this section is to compare the treemap approaches with respect to the distribution of aspect ratio values. We are particularly interested in the Slice&Dice approach as it offers high consistency.

Method, Results, and Discussion

For all systems listed in table 5.2 and all treemap approaches listed in 5.1.1 (T_{Sq}, T_{St}, T_{P_S}, T_{P_M}, $T_{P_{SS}}$, and T_{SD}) a corresponding layout is computed for the last supergraph version. Using the last version of the supergraph yields more data points because the systems typically grow during evolution. Also, the supergraph contains deleted entities as well. All treemaps are drawn into a 1×1 square rectangle. No offsets between node representations are used. The aspect ratios of all layout rectangles are computed, grouped by treemap approach, and analyzed with respect to their distribution. For each treemap approach, a plot that graphically characterizes the distribution of aspect ratios is computed.

System	T_{Sq}	T_{St}	T_{P_S}	T_{P_M}	$T_{P_{SS}}$	T_{SD}
Maximum	8237,34	9654,37	9082,81	10787,88	20946,33	318269,06
Average	8,87	8,71	10,16	18,21	16,07	1160,67
Q_3	3,46	4,44	4,56	7,01	6,20	386,14
Median	1,88	2,19	2,12	2,73	2,59	43,41
Q_1	1,33	1,41	1,31	1,48	1,47	6,37
Minimum	1,00	1,00	1,00	1,00	1,00	1,00

Table 5.9: Aspect Ratio Characteristics of Treemap Approaches

Figure 5.14 shows the aggregated results for all systems, the values are listed again in table 5.9. Each plot covers all 16 systems in their final states and thus aggregates the aspect ratios of 20.058 layout rectangles. Due to the wide range of values, the results are presented on a logarithmic scale. The following observations can be made:

- Except the minimum values, all characteristics of the distributions (median value, average value, and maximum value) are higher by at least one magnitude for T_{SD} compared to all other treemaps. Its interquartile range covers (a) larger values and (b) a much larger range of values.

- For all treemaps, the average values are higher than the median value and exceed the interquartile range.

The data confirm results of previous studies: The Slice&Dice approach yields higher average aspect ratios than the other treemap approaches. Additionally we see that the average value typically is much higher than the median value and even exceeds the interquartile range clearly. This means the high average aspect ratios for Slice&Dice in fact result from exceptionally large extreme values. The largest rectangles have an aspect ratio worse than 318.000, which is a magnitude higher than the other treemap approaches. But besides that, the distribution also shows that still 75% of all rectangles in Slice&Dice treemaps have aspect ratios above 6,37. Consequently, high aspect ratios are rather typical for Slice&Dice, they are not the exception. Beyond that, the typical aspect ratios for the Slice&Dice treemap cover two magnitudes which allows for the conclusion that usually any two rectangles in a Slice&Dice treemap are rather difficult to compare in size.

The results do not rely on the particular software systems, similar results are obtained when analyzing each system separately. Respective plots are given in appendix B.3. For all systems we see similar distributions. The Slice&Dice clearly performs worst: It yields the highest average aspect and median values, and highest and largest interquartile ranges. These distribution characteristics again are at least one, in some cases even two magnitudes above the corresponding distribution characteristics for the other approaches.

The data given in table 5.9 justifies the conclusion that the Slice&Dice approach should not be considered for software cities.

5.4 Summary

In this chapter we empirically analyzed several layout approaches (EVOSTREETS, a state-of-the-art rectangle packing, and the most popular treemaps) with respect to the layout quality criteria layout consistency and layout effectiveness. Layout consistency has been refined further into criteria concerning node positions, node neighborships, and node orderings. We quantified these aspects by corresponding measures taken from the literature and obtained a comprehensive understanding of layout consistency by analyzing each of these measures from three perspectives: A TOP PERFORMER analysis clarified which approaches perform best without taking quantitative differences into account. A DISTRIBUTION ANALYSIS allowed for conclusions about the typical behavior during evolution. Further, an analysis of the SUSCEPTIBILITY TO SMALL CHANGES provided insights to how the approaches behave for small software changes. Layout effectiveness has been refined further into the criteria layout compactness and aspect ratios which both are directly measurable and interpretable.

The results clearly show that with respect to layout consistency the EVOSTREETS approaches E_L and E_G as well as the treemap approach Slice&Dice outperform all other approaches with respect to top performance, typical behavior, and susceptibility to small changes, and with respect to node displacements, node neighborships, and node orderings. The Slice&Dice treemap additionally even offers space-fillingness which in fact would make it a good candidate for application in software city visualizations. However, high consistency and high compactness come at the price of strongly varying, extremely large aspect ratios which are not caused by a few outliers but which are rather typical for the Slice&Dice approach.

Concerning the EVOSTREETS approaches we see that the use of rectangle packing actually increases layout compactness. On the other hand, it can decrease layout consistency, but this is not necessarily the case: A simple grid based rectangle packing that balances layout compactness and layout consistency offers a similarly high layout consistency as the standard EVOSTREETS approach E_L. In contrast, the use of a rectangle packing that targets high compactness clearly decreases layout consistency.

The results also show that the rectangle packing approach P clearly falls behind all other approaches with respect to layout consistency. This is independent from which measure we used. On the other hand, the approach offers a higher compactness compared to all EVOSTREETS approaches.

E_G fulfills all requirements best: It has a high expressiveness since it encodes the structural software evolution, offers a very high consistency, preserves aspect ratios, and yields sufficiently compact layouts.

The purpose of computing is insight, not numbers.

Richard Hamming
(1915 - 1998)

Chapter 6

Thematic Software Cities

This chapter addresses the third step of the visualization pipeline described in section 1.3, i.e. the construction of thematic software cities which encode scenario specific software data. Thematic software cities are models which directly can be visualized as thematic maps and explored interactively. The thematic software cities discussed in this chapter are built on the EVOSTREETS layout approaches introduced in section 4.4. The EVOSTREETS approaches represent both the system decomposition and the structural system evolution in the layout which results in higher layout expressiveness and higher layout consistency. These particular characteristics allow for supporting additional application scenarios in the context of program comprehension and reverse engineering, quality analysis and assessment, and monitoring ongoing software development and maintenance.

In the first section, we first discuss the main application areas software cities have been used for so far and discuss the contribution of the EVOSTREETS approach to these application fields. In section two, the mapping of software analysis data to software cities to support these scenarios is described. Also, we present some cartographic methods used in this thesis to improve the efficiency of the resulting thematic software city visualizations. Finally, section three discusses results of two case studies which show the specific benefits of the EVOSTREETS approach for a variety of comprehension, analysis, and monitoring tasks. Also, selected maps illustrating interesting phenomena are discussed.

6.1 Areas of Application

Software cities have been used for two purposes: program comprehension and quality analysis. In the sequel, the contribution of the above described EVOSTREETS layout approach to these areas of application are discussed. The particular consistency property of EVOSTREETS layouts supports an additional area of application

that has not sufficiently been addressed in the past, i.e. the continuous monitoring of program evolution which is described afterwards.

6.1.1 Program Comprehension and Reverse Engineering

Many software city approaches directly address program comprehension and reverse engineering. The goal of program comprehension and reverse engineering is to gain and to document high level structural, evolutionary, and social knowledge about the project code base and the project team from available data sources like the source code, version control systems, or issue tracking systems. While nearly all software engineering activities require an understanding of the corresponding software system, this field of application for software cities is particularly relevant for project newcomers who must become familiar with a new software system, as well as for maintenance personnel who perform tasks on legacy systems or poorly documented systems and thus need to quickly acquire system knowledge. Typical goals are to gain a structural overview of the system parts, their responsibilities, and their dependencies between each other.

CONTRIBUTION
The high expressiveness of the EVOSTREETS approaches that results from the simultaneous representation of the system decomposition and system evolution supports several tasks of program comprehension and reverse engineering because it provides valuable evolutionary background on software artifacts and structures. Depicting the age of software elements by means of elevation helps e.g. in identifying structures that evolved during the same period of time, the identification of functionally cohesive substructures, and also the detection of non-syntactical, semantic dependencies which may otherwise be hidden. Examples of corresponding visualizations are discussed in section 6.3.

6.1.2 Quality Analysis and Assessment

The analysis of system quality is another traditional field of application for software cities. Quality problems are typically detected by computing software quality metrics or running dedicated analysis tools searching for structural patterns which are supposed to be problematic (e.g. bad smells as described by [70], circular dependencies, or duplicated code). These analyses yield huge amounts of data which are difficult to overview and comprehend without appropriate visualizations. Software cities have proven to support well in handling these large data sets ([192]) and nearly all software city approaches besides structural software data also include data about software quality aspects (e.g. design disharmonies [196], complexity [31], or strongly connected components [146]).

CONTRIBUTION
All previously addressed analysis scenarios are supported by EVOSTREETS based software cities as well. Design disharmonies, complex entities, or strongly connected components can be visualized in the very same way. The particular benefits of the EVOSTREETS approach are its expressiveness and consistency. Especially its expres-

siveness supports quality analysis and assessment tasks because evolutionary data is permanently visible. Thus, quality analyses can be performed in permanent consideration of the respective systems' evolution which allows for better judging quality problems, directing resources for quality assurance activities (like restructurings, or review preparations), and assessing the potential impacts and risks of software modifications. Examples of corresponding visualizations are given in section 6.3.

6.1.3 Monitoring Program Evolution

Continuously monitoring the evolution of software structures and software quality data allows for detecting quality deficiencies and potentially harmful trends like design erosion early in the development process. In contrast to the aforementioned application areas of program comprehension and quality analysis, the continuous monitoring of program evolution during ongoing software development and maintenance has not appropriately been addressed by previous research.

Monitoring program evolution must not be confused with retrospective analyses of software evolution as described in [192]. For retrospective analyses, the history of a system is typically analyzed with respect to particular analysis questions. In this analysis setting the history to be analyzed is known. Consequently, consistent software cities can easily be constructed. For monitoring scenarios, however, no such complete history is available simply because the future development of the system is not known yet.

CONTRIBUTION
The EVOSTREETS approach provides higher layout consistency in comparison to the state-of-the-field layout approaches used for software cities so far. Clearly, all software city approaches described in prior work can be used for visualizing data for evolving systems as well, and clearly visualization consistency besides consistent layouts also includes e.g. consistent mappings of analysis data as well, but the use of highly consistent layouts that serve as stable skeleton for anchoring analysis data eases the design of consistent software cities.

6.2 Thematic Mapping

So far we have only discussed spatial software models for software cities. Depending on the specific program comprehension and quality analysis tasks a wide variety of different data must additionally be represented in software cities. The process of adding these task specific data to the spatial software model is called thematic mapping. The quality of thematic mappings depends on the expressiveness, effectiveness, and consistency of artifact representations which will be discussed first in this section. Afterwards we describe approaches to depict dependencies between these artifacts, and finally we present some cartographic methods used in this thesis to improve the efficiency of thematic software city visualizations.

6.2.1 Depicting Software Artifacts

In software cities, artifacts like classes are usually displayed as buildings. Properties of these buildings typically are used to express properties of the corresponding artifacts. In their simplest form these buildings are simple boxes and their dimensions (width, depth, and height) and color are used for encoding data. More elaborated approaches ([2]) include different types of buildings (figure 6.1(a)) to distinguish between classes of different size categories and type: Houses, Apartment blocks, and Office buildings represent object oriented classes of different size whereas City halls depict $.h$-files. Finer-grained artifacts like methods are shown as Stickman.

In [192] classes can be represented as box piles (figure 6.1(b)). Each box represents a particular method of the corresponding class. As the system grows, new methods may be added to the class which are represented as boxes placed on top of the corresponding pile. On the other hand, methods may be removed from the class again. In this case, the corresponding boxes are cut out which causes the occurrence of holes as depicted in figure 6.1(c). Color can additionally be used to encode e.g. the age of methods. Consequently, the representation allows for analyzing the evolution of classes: Strongly restructured classes typically appear as instable piles of boxes with lots of holes.

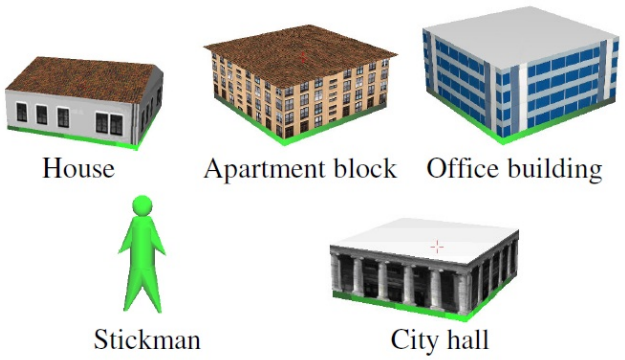
More approaches can be found in the literature, e.g. [145] who also use building categories to represent data, or [30] who use parts of houses to depict separate data in such a way that unshaped buildings point to potential quality deficiencies in the corresponding artifacts.

PROPERTY TOWERS

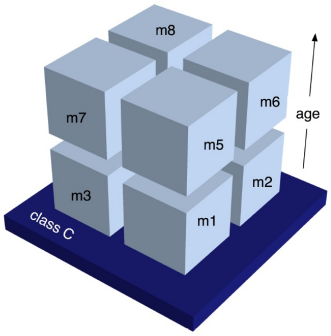
Clearly, the representations described in the literature can be used for EVOSTREETS based software cities as well. For the purpose of this work we use simple but expressive and effective representations, so-called Property Towers as depicted figure 6.2(a), for software artifacts and their properties.

Property towers may consist of a configurable number of segments (i.e. cylinders or boxes) which are stacked on each other. Each of these segments provides a height, radius, color and transparency of each segment (in the case of using boxes depth and width instead of radius) which can be used to encode specific properties of the corresponding software element. For example, the number of methods of a class may be represented by the corresponding box height, while at the same time the coupling of the class may be depicted by the base area. Finally, box color could additionally be used to encode subsystem membership.

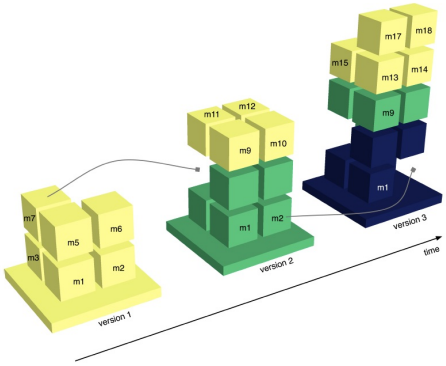
Property towers must be tailored for each application scenario. One possibility to do this is by specifying the number of segments to be stacked above each other in advance and by defining a separate mapping from element properties (like size, complexity) onto each segment's visual properties. In this case, each segment represents a particular set of element properties. Alternatively, to depict the evolution of particular properties of software elements, each segment might as well represent one particular version of the corresponding software element. Stacked above each



(a) Building Types in EvoSpaces ([2])

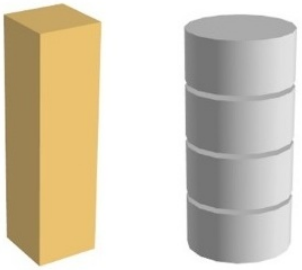


(b) Fine-Grained Representation of Classes in CodeCity ([192])

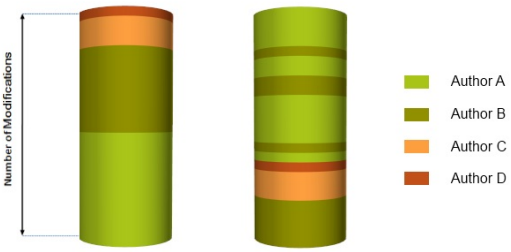


(c) Fine-Grained Analysis of the Evolution of Classes ([192])

Figure 6.1: Representations of Software Elements in Software Cities



(a) Property Towers



(b) Authorship Towers

Figure 6.2: Representing Software Elements with Property Towers

other, the property tower would then allow for analyzing the evolution of particular software aspects.

Property towers have previously been used by [190], [104], and [174] for the depiction of software artifacts, their properties, and their changes during software evolution. Depending on the specific mapping of analysis data, property towers can be used to study a wide range of very different analysis data, e.g. runtime data like execution time or memory usage ([207]), software test related data like test coverage and test success ([121]), or data about the team structure. Team structure and authorships have previously been investigated by other researches as well, e.g. [77], [189], [138], and [7]. Also several tools have been implemented to recommend authors who may have expertise on particular software entities, e.g. the *Expertise Recommender* ([137], [136]), the *Expertise Browser* ([140]), or the *xFinder* ([97]) tool. But whereas all these approaches characterize authorships quantitatively, visualizing authors in software cities may provide a qualitative understanding of authorships.

To depict authorships in software cities, so-called Authorship Towers are used. Authorship Towers, as depicted in figure 6.2(b), show modifications made by a selected set of authors for each software element. For this purpose, they consist of several colored segments each of which indicates an author-specific modification of the respective element. The color of each segment is used to encode the author who made the respective modification. Authors can be selected interactively; modifications by authors not contained in this selected author set are represented by transparent segments. Thus, the total height of each authorship tower always represents the number of all modifications applied to the corresponding software element.

There are two possibilities to order the segments: First, they are sorted by author, i.e. all segments for modifications performed by the same author are stacked to form one composite segment for that author (left tower in figure 6.2(b)). Second, segments are sorted in ascending temporal modification order and displayed bottom up, i.e. segments for old modifications are displayed at the bottom whereas new modifications are displayed on top of the tower (right tower in figure 6.2(b)). Whereas the first kind of ordering supports to rate the familiarity of authors with particular software elements, the second kind of ordering is particularly useful to detect drifts in code ownership

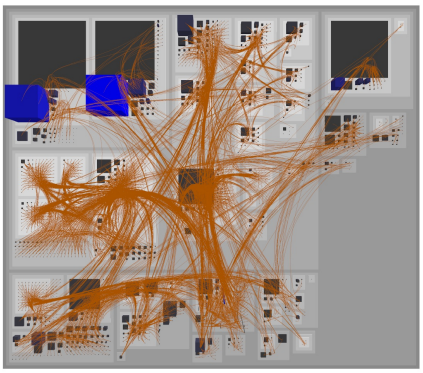
Kerstan discusses an interesting variation of property towers in [98]. Rather than using simple geometric shapes like rectangles or cycles to depict one or at most two properties at the same time, Kerstan suggests to use Kiviat diagrams instead which allow for depicting an arbitrary number of properties. Piled above each other for each version, the resulting property towers would allow for analyzing the evolution of several software properties at the same time.

6.2.2 Depicting Relations

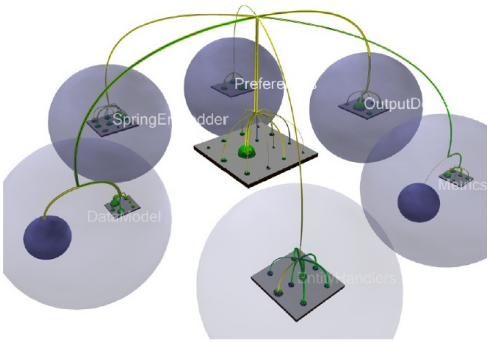
Besides structural system decomposition and analysis data, the comprehension of dependencies between software artifacts is an essential task for many scenarios.

The visualization of dependencies can support in gaining an understanding of the overall coarse grained dependency structure, in detecting violations, or in assessing potential impacts of modifications to particular software artifacts.

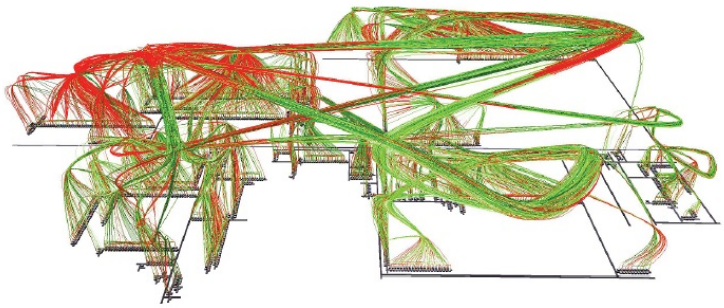
Depicting relations on coarse abstraction levels is still an unsolved problem for software cities but for other visualization approaches as well. Clearly, drawing straight lines for each dependency soon causes intersections and visual clutter such that neither individual dependencies nor streams of similar dependencies can be identified. In general, the goal is to obtain short routes without edge intersection that allow for recognizing coarse-grained dependencies. To solve this problem, the visualization of large numbers of dependencies typically uses routing and/or bundling techniques. Examples of such are given in figure 6.3 which shows three solutions described in the literature for visualization of relations in software city and landscape visualizations.



(a) Relations in CodeCity ([192])



(b) Relations in Software Landscapes ([16])



(c) Relations in Street-based Cities ([41])

Figure 6.3: Depicting Relations in Software Cities and Landscapes

In contrast to previous software cities, EVOSTREETS based approaches provide an explicit city infrastructure based on streets that can be used to route dependency representations similar to traffic in real cities. An example is depicted in figure 6.7 (p.133). When routing edges on the basis of the street system, we are using right-

hand traffic to indicate dependency directions. The use of color would additionally increase the readability. This kind of representation implements a rather strong edge bundling. It is, however, most useful for exploring particular dependencies between entities rather than coarse-grained system dependencies and thus we use them as supplementary technique, only.

As figure 6.3 illustrates, it is still very difficult to decipher the overall dependency structure on the system. One reason for this may be that the positioning of elements in the city does not bear any relational meaning. In the CodeCity approach, for example, leaf nodes are placed to optimize the overall layout with respect to compactness. This optimization is done without regarding structural dependencies between software elements. The same is true for Software Landscapes described by [16] and the EVOSTREETS approach. In chapter 7 we discuss an approach, i.e. software city landscapes, that allows for depicting coarse grained structural dependencies in addition to the structural and evolution data represented by EVOSTREETS based software cities.

6.2.3 Cartographic Methods

To further enhance the effectiveness of our thematic software city visualizations, two more visualization techniques are used in this thesis: Contour lines increase the readability of landscape elevation levels. Orthogonal projections yield two dimensional maps without occlusion.

Contour Lines

Terrains show element elevations and the overall elevation structure which, however, would hardly by recognizable without the use of means like terrains. Thus, adding a terrain substantially increases the readability of the visualization. On the other hand, terrains do not very well support a comparison of heights at distant regions of the visualization because the elevation level is not readable in terms of concrete elevation values. To overcome this shortcoming, we make use of contour lines as displayed in figure 6.4. Contour lines allow for an approximation of element elevations by simply counting the number of contour lines below that element. Hence, elevation levels in distant regions can easily be compared by the number of contour lines surrounding them.

Contour lines can be used at arbitrary elevation levels. In the EVOSTREETS approach, landscape elevation in general indicates the creation time of software entities in terms of versions stored the software model. Therefore, we represent each of these versions by a contour line. If, for example, the software model is populated on a daily or weekly basis then each contour line represents a corresponding day or week, respectively. As a result, new elements are surrounded by only a few contour lines. The older elements become the more contour lines surround them. Since counting contour lines quickly becomes more difficult as the number of lines in the visualization grows, different line widths should be used to highlight any n^{th} contour line. In figure 6.4, for example, each 5^{th} version of the software model is

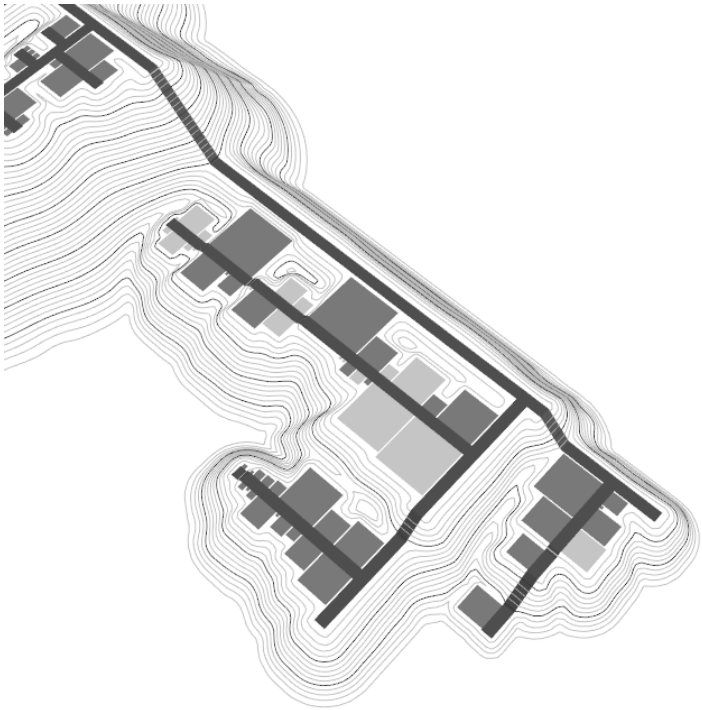


Figure 6.4: Depiction of Elevation Levels by Contour Lines

represented by a thick contour line whereas all other versions are represented by thin contour lines.

Orthogonal Projections: 2D Maps of Software Cities

Software Cities are three dimensional spatializations of software systems, and most approaches also use all three spatial dimensions to encode data. Hence, a comprehensive analysis of this data requires perspective views or interactive exploration of the software city. Even though such complex 3D/2.5D visualizations can be powerful tools (especially when visualizing inherently 2.5 dimensional structures like cities) they are not necessarily the most adequate instrument for each application scenario. Especially for simple overview, illustrating, or navigational purposes 2D visualizations may be easier to read, comprehend, and use than complex 3D visualizations since they avoid problems caused by occlusion and perspective distortion.

2D visualizations can be derived from 3D visualizations by projections as displayed in figure 6.5. The resulting city map depicts the overall structure of a system and serves as a skeleton to locate analysis data, for example, to highlight development hotspots or test-coverage. While in their simplest form city maps do not provide information about elevation and the height of property towers, the additional use of cartographic techniques as contour lines or shadows can reveal this information even 2D city maps. Figure 6.5 gives an example 2D map that is enhanced by both techniques. In this map both elevation and property tower height are directly en-

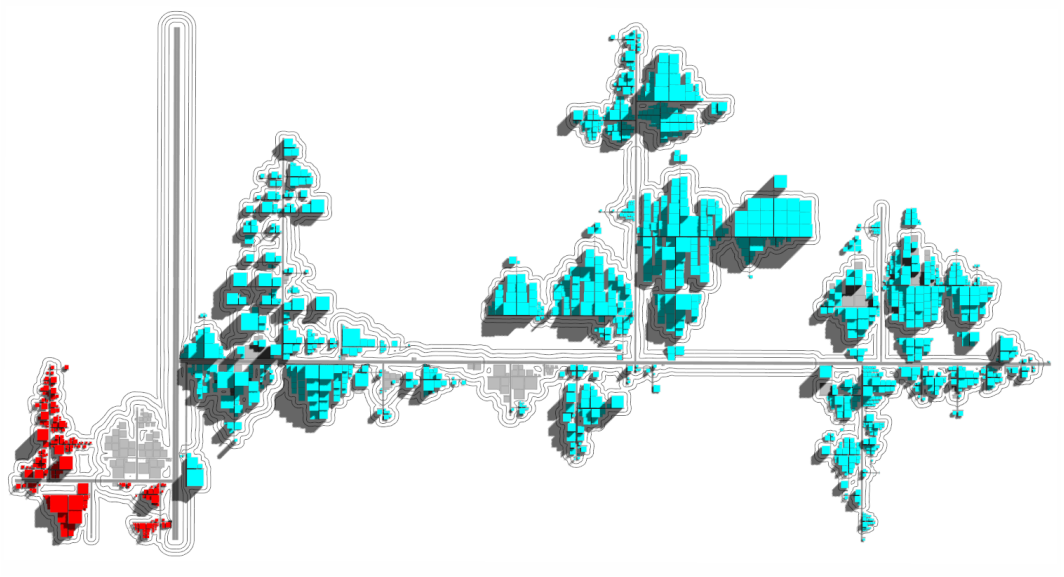


Figure 6.5: 2D-Projection using Contour Lines and Shadows

coded and readable.

6.3 Case Studies

Using the mechanisms described above, we evaluated the EVOSTREETS approach for several systems. First, we analyzed our visualization tool CrocoCosmos because we know its internal structure, evolution history and the organization of the team that developed it. Second, we applied our visualizations to two commercial software systems and obtained feedback from the responsible industrial partner. Third, we visualized all open source projects listed in the appendix with standard thematic mappings to obtain a qualitative understanding of the strengths and weaknesses of the approach. The results of these case studies are described in this section.

6.3.1 CrocoCosmos

CrocoCosmos is a long term research project at the Software and Systems Engineering Research Group at the Brandenburg University of Technology (BTU, Germany). It covers approximately 12 years of active development. CrocoCosmos has always been a software visualization tool, and though its specific goals ranged from the exploration of software quality analysis data represented as interlinked html document sets to the visualization of software systems as software cities. Although not the whole history is covered by the visualizations presented here (we look at the last three and a half years only), there are legacy parts in the codebase that are wired with the current system but which are not used or maintained any longer. To a large part CrocoCosmos is developed by students who in the context of their student

theses work for a short period of time on typically small parts of the codebase. CrocoCosmos is written entirely in JAVA, thus instead of using the generic term *system elements* we use JAVA specific terms *package* and *class* when discussing particular results for CrocoCosmos.

Understanding Authorships

The analysis of authorships with EVOSTREETS based software cities supports comprehension and reverse engineering scenarios because it allows for identifying functionally cohesive subsystems developed by particular authors during a short period of time. Figure 6.6 shows the authorship map for our example system. As discussed above, the height of the authorship towers represents the number of modifications that were applied to the corresponding class in the past. The radius of each tower represents the number of ingoing and outgoing dependencies.

The contributions of the main developer A are shown in dark green. From the early days of the project until now he is the author and main contributor of many classes throughout the system. In contrast, the developers B (dark brown) and C (light brown) are authors of more recent extensions of the system. Both are specialists for local and clearly separated parts of the system. Developer D (yellow) was the author of an early extension of the system which also required some adaptations in many existing classes. This developer was active for a certain period of the development history, only.

The authorship map shows that developer C added two different system extensions depicted at the bottom right and the bottom left of the software city visualization, respectively. To analyze whether these extensions depend on each other, all structural dependencies of both can be added interactively. Figure 6.7 shows all structural dependencies for those two subsystems that were developed by the light brown developer. Obviously there is no direct structural dependency between these subsystems. Instead they are related to some old classes developed by developer A.

The reason for this is that developer C first developed a new rectangle packing layout approach and added this implementation to the layout subsystem at the lower right region of figure 6.7(a). The results of this computation (i.e. the positions of all rectangles) are stored into the central data structure which has been part of the project from its very first days. Later on he developed an analysis subsystem to evaluate quality properties like consistency and compactness of his new layout approach and added this implementation to the analysis subsystem at the lower left region of figure 6.7(b). For this purpose, all rectangle positions computed by the new layout approach must be read from the central data structure again. This created an indirect data dependency between the layout computation subsystem and the layout analysis subsystem.

While this example very clearly points to a non syntactical data dependency among two subsystems hidden dependencies certainly cannot be detected as clearly as shown above for the general case. This is especially true since programmers usually work on several distinct and independent parts of the software. Thus, when study-

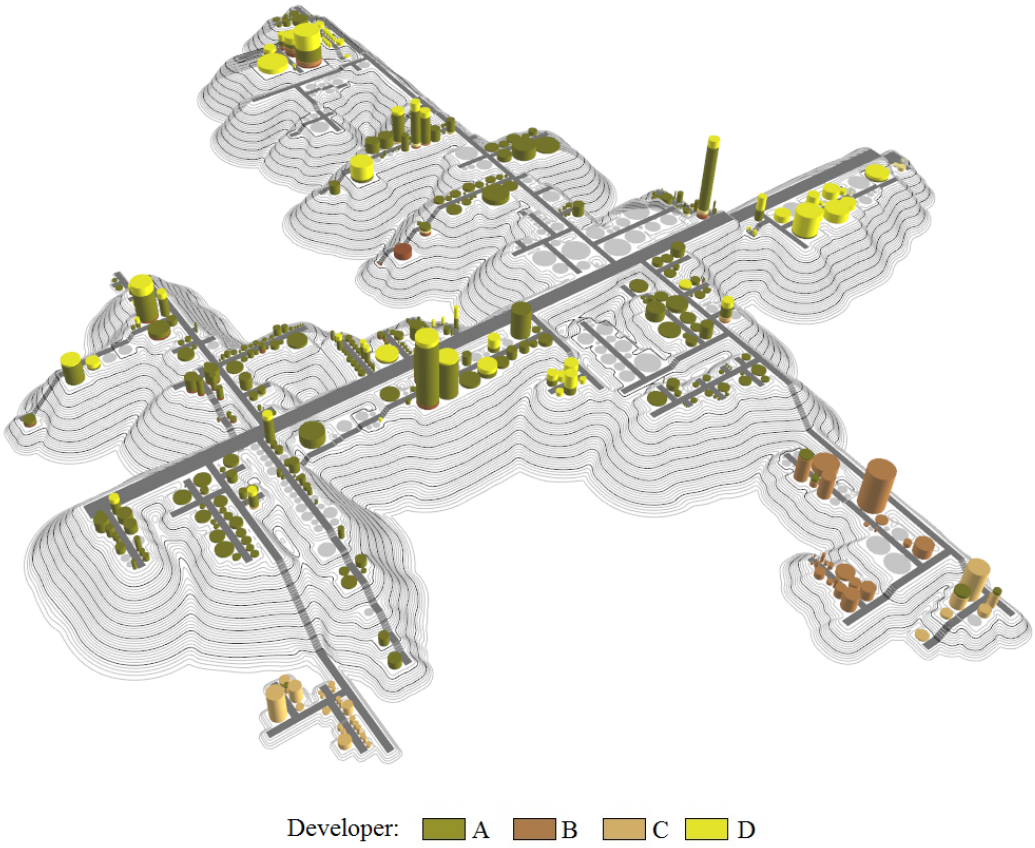
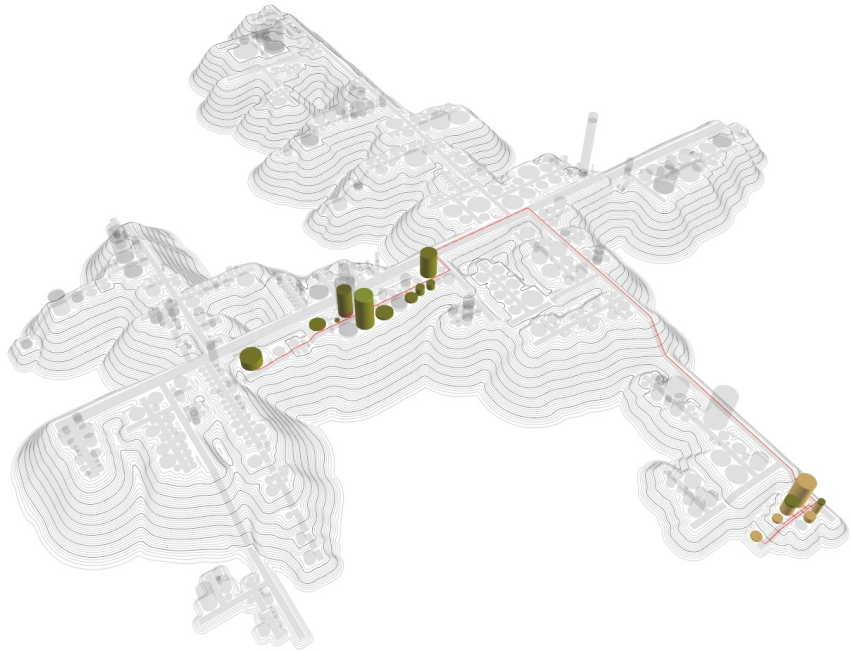
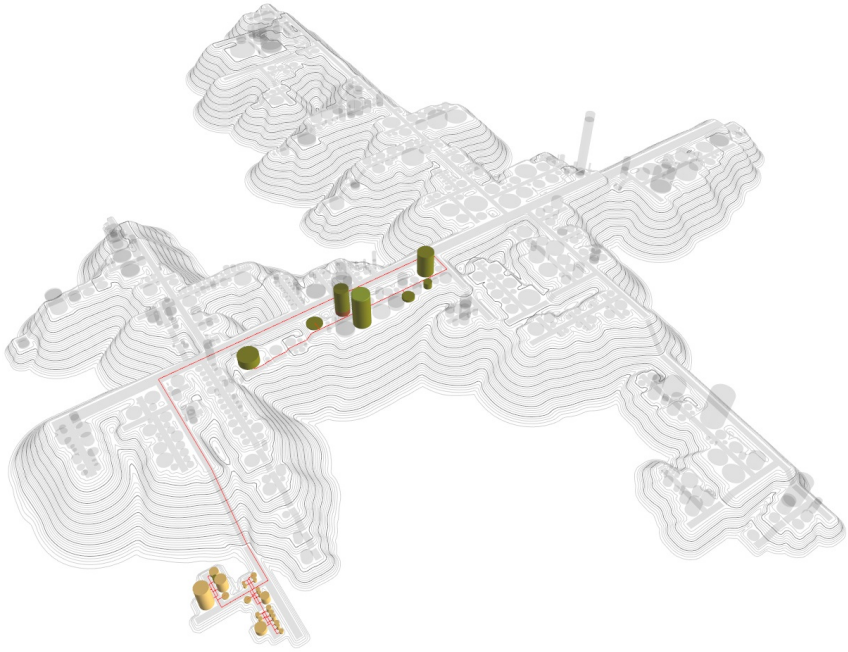


Figure 6.6: Authorship Map for CrocoCosmos



(a) Dependencies between computation and data structure



(b) Dependencies between data structure and analysis

Figure 6.7: Dependencies of an Author's Subsystems

ing authorships to detect such dependencies it might be reasonable to consider only those modifications of a particular author that were performed during a relatively short period of time.

Analysis of Modification History

Version control systems offer a huge amount of data about the evolution of software systems. Besides authorships and creation time (which is already represented in the city layout), additional data regarding the system evolution (like the version when an element was modified last or the number of modifications that were applied to an element in the past) can easily be extracted and represented in the software cities as well. To analyze these data, we map the number of modifications of each artifact to the corresponding tower's height. Additionally, the version when a software element was modified last is mapped to the corresponding tower's color using a color map as depicted in figure 6.8. As a result, software elements which have not been modified for a long time appear as blue towers whereas software elements that were modified very recently are depicted by yellow towers.

Figure 6.8 shows the modification history map for the CrocoCosmos system. By using this map one can easily distinguish between different situations: (1) Flat blue towers on high elevation levels, often located on separated plateaus, represent classes which have been part of the system for a long time, which have barely been modified after their initial creation, and which have not been modified for a long time. There are two possible reasons for such a situation: Either these classes implement very stable functionality, or their functionality is not used any longer and, therefore, they are not maintained at all. In this case these classes and possibly even their containing packages could be candidates for being removed from the system. The small blue towers highlighted in figure 6.8 depict classes of the analysis subsystem already discussed above. They implement different graph analyses to e.g. detect circular dependencies. The author that is responsible for these analyses left the project long before the evolution history that is depicted in figure 6.8, thus he is not represented in figure 6.6. Instead, figure 6.6 shows that developer A modified these classes. This, however, results from a transition of the project's codebase from one version control system to another which was done by developer A. After this transition, the classes have not been modified anymore.

(2) In figure 6.8 another interesting situation is highlighted: Tall yellow towers on high elevation levels represent old classes that have been part of the system for a very long time. They have been changed a lot since their initial creation and were modified very recently. Towers like these may indicate potential design problems and should be subject to further investigation as, for example, manual inspections. The towers highlighted in figure 6.8 point to classes of the central data structure of CrocoCosmos. We analyzed the reasons of those frequent changes of the corresponding classes by manually examining the log messages stored in the version control system. For the most frequently modified class we found out that only 32 of all 67 modifications of that class included the addition of new functions. The other 35 modifications can (on the basis of their commit messages) be classified as bug

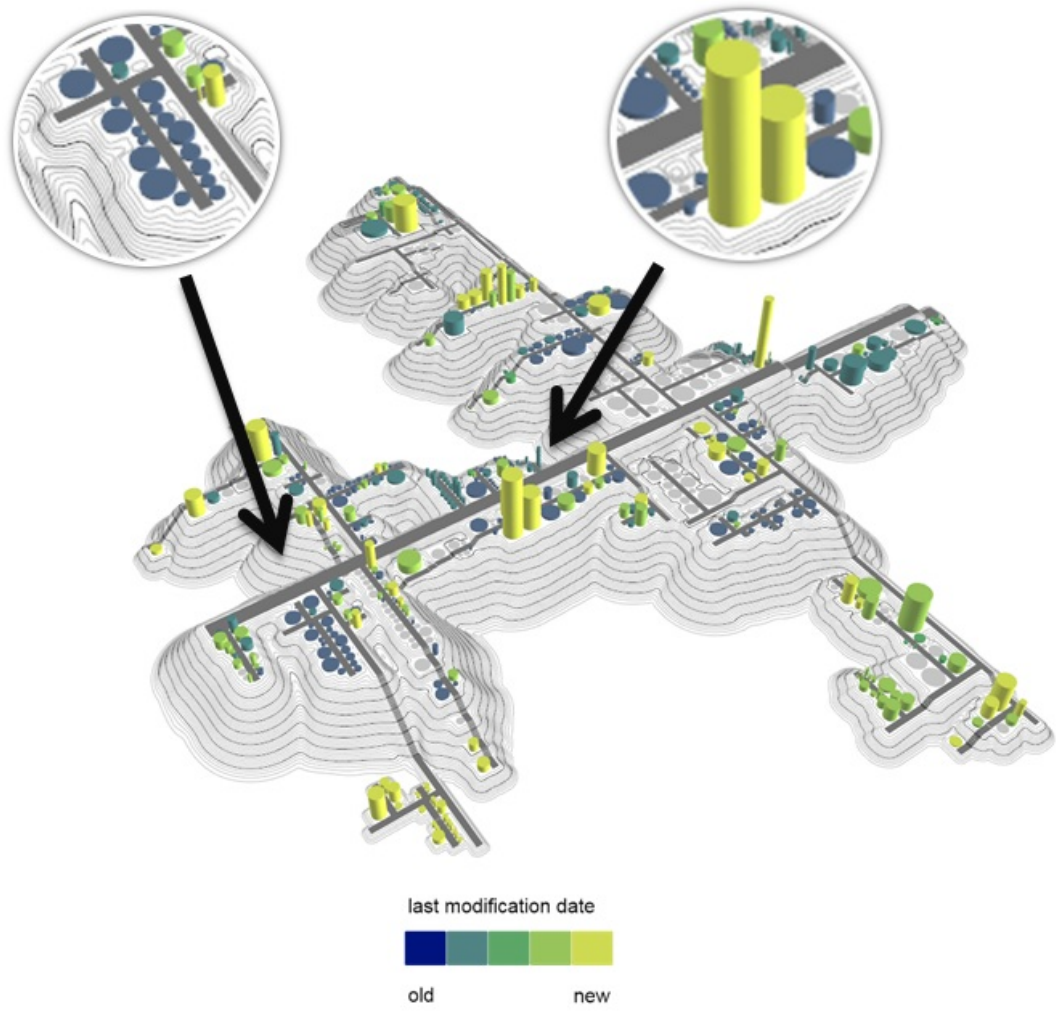


Figure 6.8: Modification History for CrocoCosmos

fixes or code improvements (e.g. general code cleaning or refactorings). Thus, in the past much time had to be spent solely for correcting and improving the class. The evolution history map for CrocoCosmos thus helps in identifying candidates for further quality assurance activities very quickly.

Analyzing Coupling Metrics

Another thematic software city for CrocoCosmos is shown in figure 6.9. Each class is represented by a tower that consists of two segments. The height of each segment represents the incoming (blue) and outgoing (yellow) structural dependencies of the corresponding class, i.e. the number of other classes using it or being used by it. As the base area of each tower again is proportional to the number of dependencies (ingoing and outgoing), the coupling property is strongly emphasized in this type of visualization.

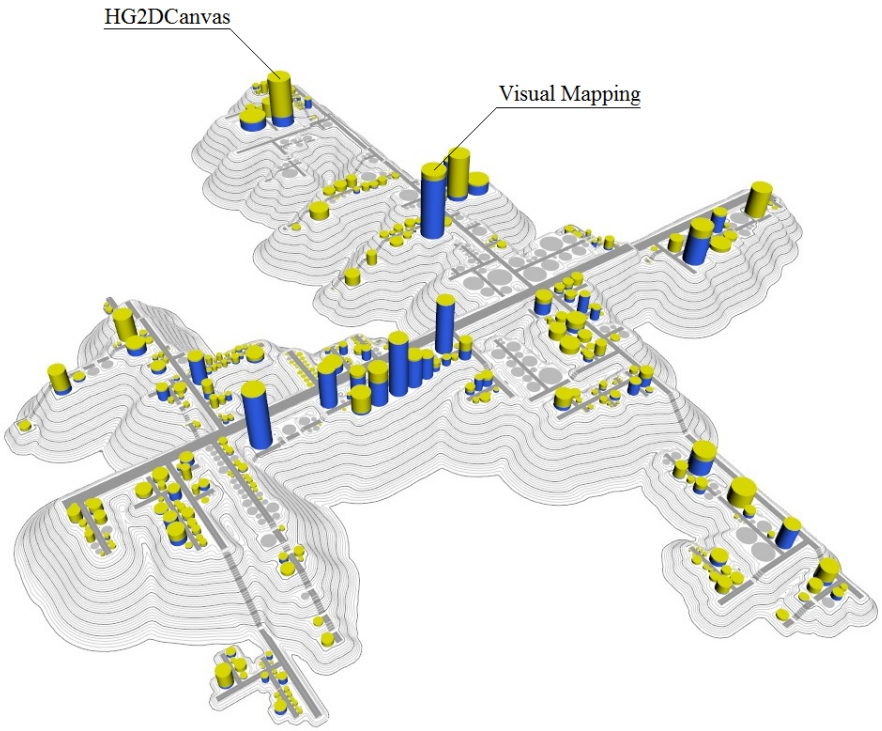


Figure 6.9: Visualization of Coupling Metrics for CrocoCosmos

The visualization clearly shows one package in the center of the city which contains several old classes with a very high incoming coupling. As already discussed above, these classes implement the system's central data structure which is accessed throughout the system. Because these classes are old and strongly coupled, any modifications carry a high risk of being incomplete and faulty, and potentially influence a large number of dependent classes that are distributed throughout the system.

Additionally, several other classes with a high coupling can be detected outside the central data structure. Class `VisualMapping`, for example, is responsible for mapping analysis data that are stored in the central graph structure to visual properties like shape, size, or color of the corresponding node representations. This class, hence, implements the thematic mapping step of the visualization pipeline discussed in chapter 1. Its results (i.e. the actual colors etc. of nodes) are directly accessed by several different visualizations (2D matrix views, 3D software cities, and many more) that are implemented in several other classes. Class `HG2DCanvas`, for example, implements such a specific 2D visualization, it allows for a visualization and analysis of particular graph structures (hyperedges). The large yellow fraction of the corresponding tower indicates that `HG2DCanvas` makes extensive use of other classes. It is, however, not strongly used by other classes which is indicated by the small blue fraction of the tower.

Complex Analysis Scenarios

Each of the thematic software city visualizations described above concentrates on one specific aspect of the underlying software system (like authorship, modifications, element coupling etc.). These maps can easily be combined with other maps to support more complex comprehension and analysis tasks. An example of such a complex analysis scenario is the identification of old software elements with high coupling that have not been modified for a long time and for which expertise is missing because the corresponding authors left the project. Such critical system components can easily be identified by switching between the corresponding thematic maps. For CrocoCosmos, we have seen exactly this situation. There is a strongly coupled old system core which has often been modified in the past, which still is often subject to change, and for which expertise is provided by one particular author only.

Besides that, the approach allows for filtering the visualization e.g. to show only those classes that were modified by a given author or in the recent past, or to show classes which were executed by a given test suite. In [121], Legde proposes a concept for visualizing test data in software cities. Test related data are represented by two specific property towers. The test coverage for a class can be represented by two segments indicating the number of lines of code that are covered or uncovered, respectively, by all test suites. Test suites, i.e. essentially classes that implement test cases, are represented by mapping the number of successful and failed test cases onto both segments in a similar way. To distinguish regular classes and classes that implement test suites, different shapes of towers (cuboids and cylinders) are used. Filtering the set of represented classes on the basis of such test data in other thematic maps helps to focus on particular system parts, e.g. those which are not tested sufficiently.

In the same way, run-time data collected during the execution of particular system functions (e.g. execution time, memory usage) can be used as filter for thematic maps. This allows (for instance) for easily combining information about error prone or computationally expensive executions with the previously discussed results from the modification history maps or authorship maps. Besides that, Zierenberg ([207]) proposed a concept that allows for visualizing such runtime data directly by mapping it to visual properities of corresponding property towers. For Java systems, this can even be done at runtime ([207]). For this purpose, CrocoCosmos uses online code instrumentation to record execution data and integrates these data into the software model at runtime. Consequently, the runtime characteristics of a software system (e.g. aggregated memory usage and computation time of classes) can directly be monitored in software cities during the program's execution.

As discussed in section 6.1, EVOSTREETS based software cities contribute to three coarse areas of application, i.e. program comprehension and reverse engineering, quality analysis and assessment, and monitoring of program evolution. Uhlig ([183]) proposed an approach that supports the first two areas of application by visualizing concepts in software cities. The extraction of concepts from the codebase is done using information retrieval techniques. The visualization of concepts in

EVOSTREETS based software cities is based on a technique described in [38] for the visualization of areas of interest in software architecture diagrams: colored ground areas surrounding software city elements are used to indicate that a particular concept is implemented by the corresponding software element. Areas representing the same concept are connected and thus form a cohesive shape. The visualizations support (especially new team members) in gaining an understanding of the concepts and their distribution in the system. Since this concept visualization technique can be combined with the analysis data described above the obtained EVOSTREETS software city visualizations can easily be used to detemine e.g. the age of particular concepts, frequently and/or recently modified concepts, concepts with a large author set, insufficiently tested concepts and much more.

6.3.2 DP-TTPB, DP-AN

We conducted a study with an industrial partner, a large logistics company. The focus of this study was not on program comprehension and reverse engineering, but on exploring the capabilities of the approach particularly for the analysis of software quality and monitoring. Thus, on the company side the study involved the quality assurance team which is responsible for assessing and monitoring the quality of externally developed code. The team already used several software analysis and visualization tools. Among others, they used the CodeCity ([192]) tool.

The study included two software systems which were analyzed in separate iterations. During the first iteration we looked at a medium sized Java system (DP-TTPB) with 354 Java classes in five major project releases. For the second iteration we used a larger system (DP-AN) with about 1500 classes again in five major project releases. For both systems, we produced visualizations depicting analysis data that is actually monitored by the quality team (standard software metrics like lines-of-code (LOC) and complexity as well as company specific metrics to detect risky code and unfinished code) and returned these visualizations including a brief textual description for each of them. We discussed the visualizations with the quality assurance team and obtained the following general feedback and project specific results.

General Feedback

From a representational point of view, the team pointed out the aesthetics and readability of the visualizations, though the latter still should be improved: On the one hand, the readability of the hierarchical system decomposition represented by the city layout was highly appreciated. On the other hand, however, it would be rather difficult to read the elevation information of the landscape provided via contour lines. In part this may be due to the small number of versions we looked at, but nonetheless it also suggests that additional support mechanisms as e.g. the use of cartographic height coloring or labeling may be necessary.

We provided both 2D and 3D maps, and an interactive tool (the 3D Scene-Explorer[1]) that allows for exploring 3D scenes for each thematic software city. Since in 2D maps

[1]The 3D Scene-Explorer is a reduced version of the CrocoCosmos tool.

building height is not visible, we included building shadows to indicate high buildings. We asked for an assessment and the users preferences. Interestingly, 3D maps were assessed more valuable as they would allow for identifying correlations between quality attributes more easily. Occlusion occurred but it would be no problem for two reasons: First, from the quality assurance perspective interesting parts are usually represented as higher buildings. Second, if the visualization can interactively be viewed from different angles then occluded buildings could easily be detected by rotating the scene once.

The 3D Scene-Explorer tool we provided for interactive scene exploration offers a rich set of navigational features like Pick-&-Zoom, Pick-&-Rotate, Fit-To-Screen, and a map mode which displays a parallel projection of the represented city is displayed. During the feedback sessions the team repeatedly highlighted the value of these interaction techniques that go beyond standard panning and zooming techniques and stated these were crucial for effectively and efficiently analyzing the 3D scenes. On the other hand we observed serious trouble in the team to find appropriate computer equipment for running the 3D analysis tool at all. Most of the computers had no sufficiently good graphics card.

Quality Analysis and Assessment

We mapped several analysis data computed with software analysis tools to the height of the buildings. In each of these visualizations we always identified the same areas of zero-height buildings (figure 6.10) distributed throughout the city for those scenarios where typical quality data like bug numbers were mapped onto the building heights. At first we supposed this to be an analysis blind spot caused by a faulty configuration of the quality analysis system and asked for the reasons. But in fact this blind spot is intentional as it points to test code and generated code which are not monitored by analysis tools. We offered to remove this code from the visualization but the team explicitly rejected this idea as it would be important to know about this code and how it is distributed in the system. However, it would be rather helpful for distinguishing hand written code, generated code, and test code representations from each other, for example by using additional means like shape or surrounding ground color.

Besides the detection of quality blind spots, the quality assurance team (not surprisingly) confirmed the results of [192] that by the use of the city visualization it would be easier to quickly gain an overview of potential quality deficiencies in comparison to standard data tables and diagrams.

Monitoring Program Evolution and Quality

One of the responsibilities of the quality assurance team is to monitor the evolution of so-called risky code and unfinished code. Avoidance of risky code has explicitly been stated as a quality requirement for both systems, and the number of occurrences of unfinished code should decline to zero at least until the final project revision gets reached. Both qualities can numerically be quantified by specific metrics

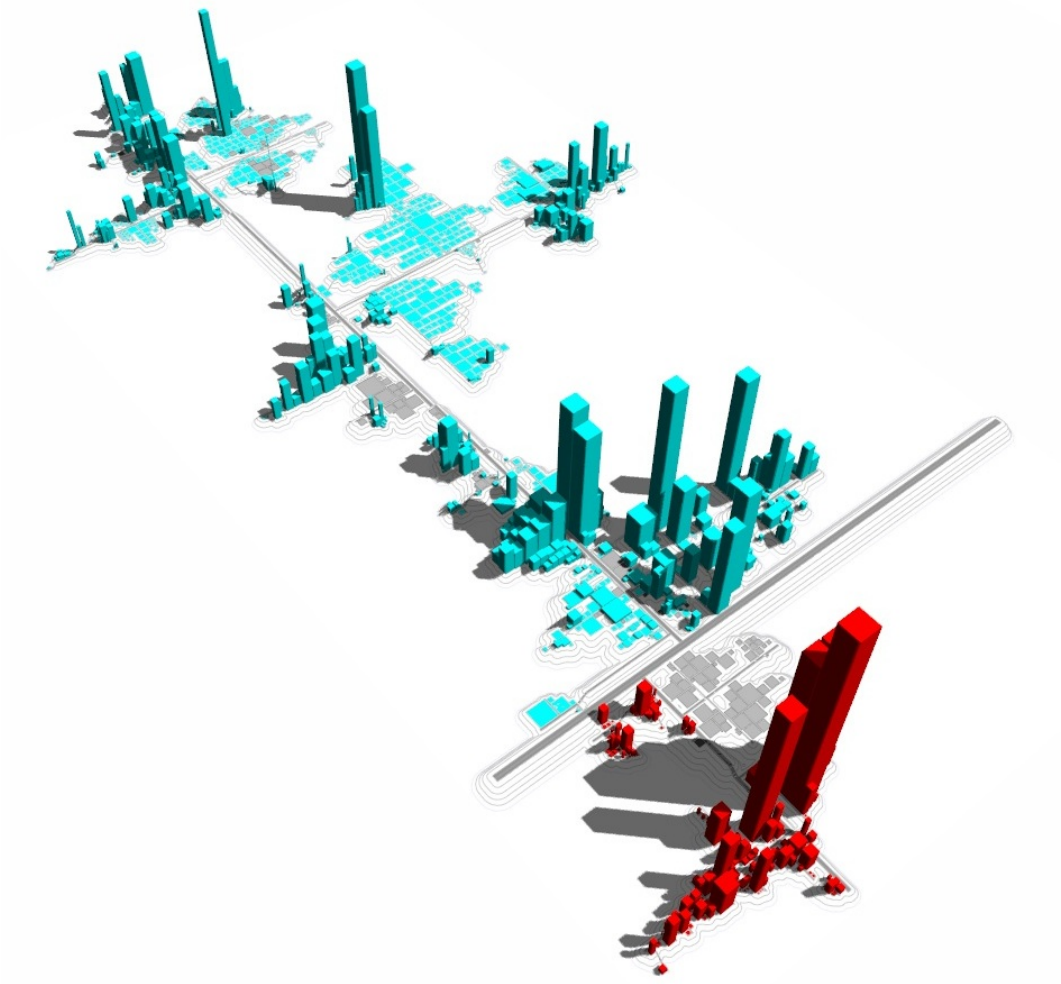
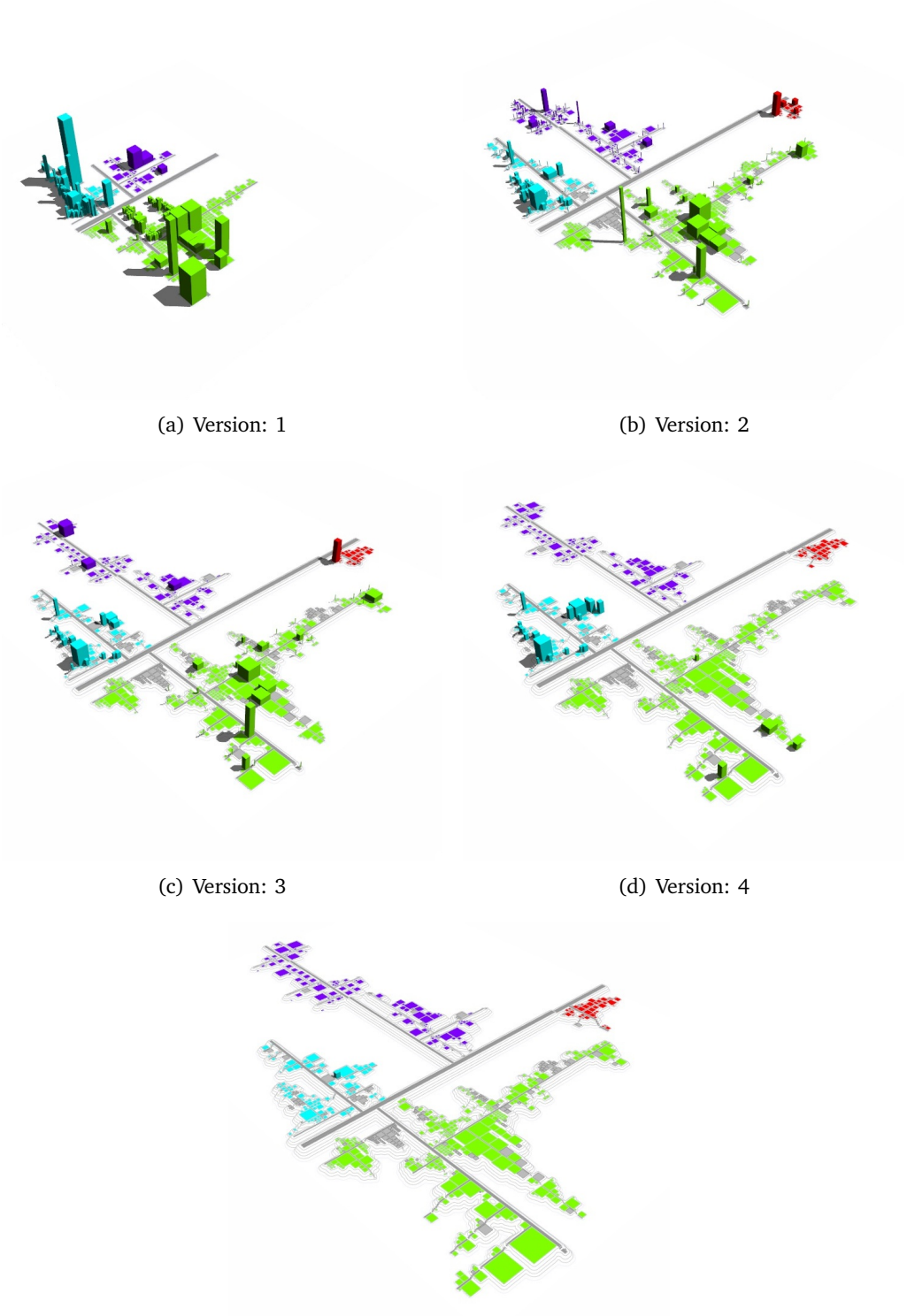


Figure 6.10: Detection of Quality Blind Spots for DP-AN



(a) Version: 1

(b) Version: 2

(c) Version: 3

(d) Version: 4

(e) Version: 5

Figure 6.11: Evolution of Unfinished Code for DP-TTPB over 5 Versions

used by the quality assurance team. We obtained the respective data, and derived respective visualizations to support this customer requested scenario during the second iteration. Again we provided static visualizations for several scenarios including the additional risky code and unfinished code scenarios and discussed them in an interview session.

Figure 6.11 shows the software city visualizations for analyzing the yet unfinished code of the project. The height of each building indicated the value characterizing the *unfinished code* property. The visualizations for both the analysis of unfinished code and the analysis of risky code, respectively, largely matched the expectations of the team: risky code has been very rare throughout the whole evolution, and the unfinished code continuously decreased until the final version has been reached. Particularly strongly coupled classes were finished rather soon which is directly visible by means of building area.

Clearly, these data can be visualized by other means like data tables and diagrams. However, the quality assurance team explicitly pointed out that the major advantage of using these visualizations is that it takes only a few moments to obtain an overview of risky and unfinished code parts, their distribution throughout the system, and their development compared against the previous revision. It'd be very much like "watching a movie" when clicking along the versions and seeing (e.g.) unfinished code continuously decrease to zero. This in fact is a quality that results from the high layout consistency of the underlying EVOSTREETS approaches.

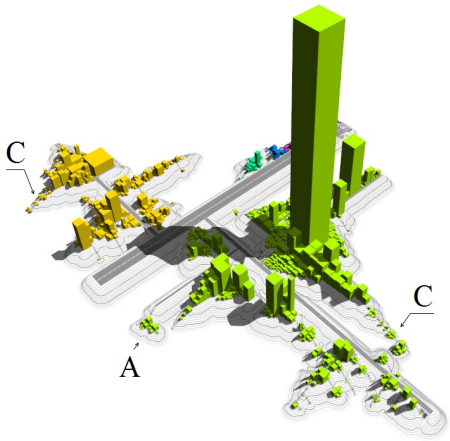
6.3.3 Selected Maps

In this section, we discuss selected maps for several open source software systems. Figure 6.12 gives an overview of the corresponding software city visualizations. All cities use the E_G EVOSTREETS approach (i.e. a simple grid based rectangle packing is used).

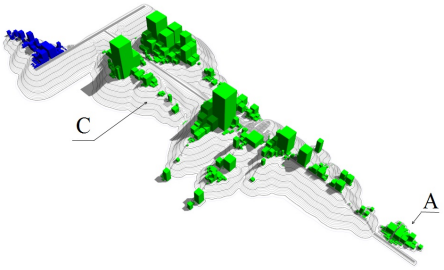
Figures 6.12(a), 6.12(b), and 6.12(c) illustrate two aspects: First, visual patterns are not specific to our CrocoCosmos tool, but they show up in each of these cities. We can find suburbs (A) indicating system extensions, high plateaus (B), mountain slopes (C) pointing to continuously growing subsystems, and disused sites (D) in these software cities. Second, although these layouts depict up to 1494 classes developed over several system versions the layouts do not suffer from low compactness. Quite contrary, the degree of separation between districts even supports in reading the hierarchical system structure.

Figure 6.12(d) depicts the authorships for the JME system. There are several developers that strongly contribute to roughly every class in the system. Besides, there is no region in the visualization that is strongly dominated by a particular author. Clearly, this organization of changes points to a collective code ownership whereas the visualization for CrocoCosmos (6.6) depicts to some areas in the system which are clearly owned by particular authors.

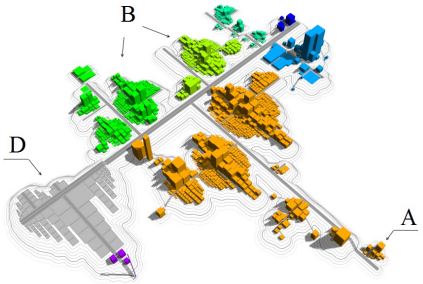
For PMD (figure 6.12) we observe that the software city visualization is dominated



(a) Derby Engine (13 versions, 1494 classes)

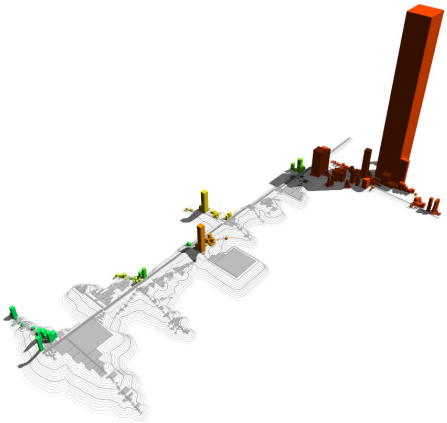


(b) Neo4J (26 versions, 561 classes)

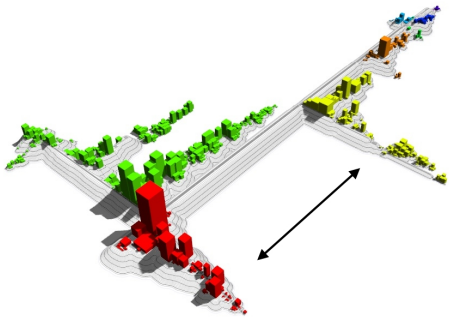


(c) OpenSAML (8 versions, 1410 classes)

(d) JME Collective Code Ownership



(e) PMD (15 versions, 687 classes)



(f) Ant (22 versions, 1044 classes)

Figure 6.12: Selected Maps

by an enormous amount of disused sites. Obviously, during its development many parts of PMD have been removed from the system or they have moved to other subsystems. Since the visualization does not tell what happened to these structures, we inspected them manually and found out that many of the elements that were originally positioned on the now empty plots actually moved to the upper right subsystem. An example is the huge tower at the upper right which originally was positioned on the large empty area on the lower left corner. Consequently, the city for PMD requires much space for depicting old removed structures whereas the actual system containing only 687 classes is widely spread throughout the city.

In case that restructurings could be detected automatically or recorded manually, one could leave a message at the ruins stating a "”We have moved.“ and possibly display a route to the new location when moving the mouse over the area. While this is an illustrative anecdote only, it actually points to another field of application for software cities in general. With the help of appropriate repositories, these visualizations could be used for project documentation purposes. Information about restructurings or refactorings could be left at the corresponding ”place“ including additional data such as the date, author, and rationale of restructuring. Otherwise, this data would have to be extracted from e.g. commit logs of the corresponding version archive or from additional project documents.

Figure 6.12(f) points to a still open compactness problem of the EVOSTREETS approaches which causes the large distance between the red and the yellow subsystem of the Ant software system. Ant initially consisted of the red and the green top level packages depicted at the lower left end of the central road. At some point in time, new top level packages were added and attached to the far end of the central road. These are depicted as a yellow district and some smaller ones beyond. In figure 6.12(f) we observe a large distance between the red and the yellow subsystems which is caused by the green subsystem located to the other side of the main street. Each street is built up of segments representing particular periods of time. The segment length depends on the maximum size (width) of the substructures branching off independent of the size these substructures are located. The large gap between the red and the yellow disticts results from the large difference in width between the green and the red subsystem.

Figure 6.13 depicts 13.991 classes of the Java Development Kit 1.6 (JDK). The city is computed using the E_L EVOSTREETS approach. Although the layout does not use any rectangle packing algorithm to improve layout compactness the city provides a clear overview of the main JDK components and allows to easily identify the largest classes (high towers). On the other, it is rather difficult to assess the ground areas in this visualization. While the figure 6.13 clearly illustrates the scalability of our approach it also indicates that we reach the upper size up to which these cities can be used for fine grained analysis and monitoring scenarios. When analyzing and monitoring tens of thousands of classes and representing. At the scale of thousands of classes, typically more coarse grained analysis data, or particularly filtered and aggregated data is of higher interest than fine grained class specific data. An example of such coarse grained scenarios is the analysis of software architecture and the

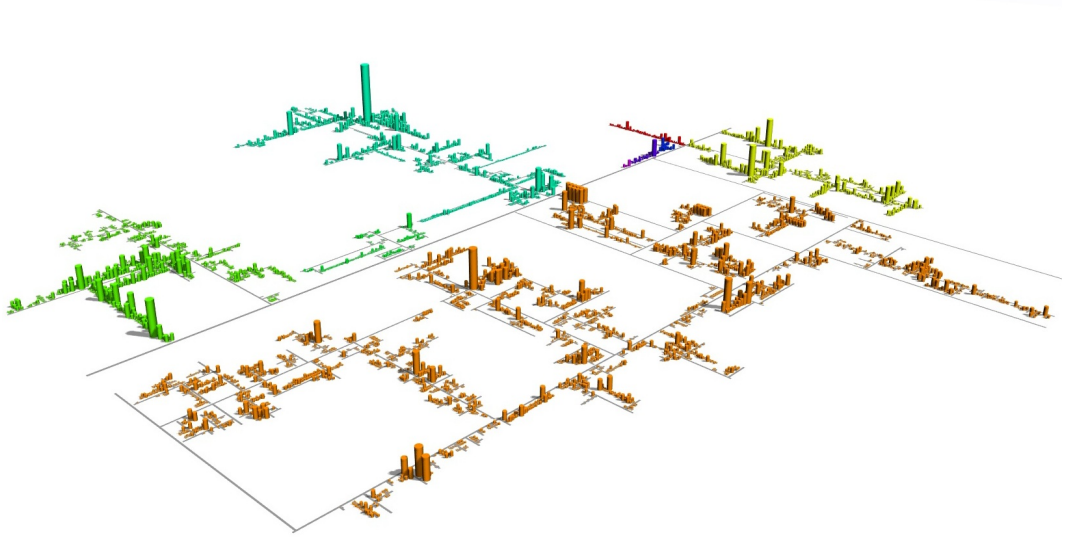


Figure 6.13: 14000 classes of JDK 1.6

early detection of architectural erosion.

6.4 Tool Support

All visualization concepts presented in this theses are implemented in our visualization tool CrocoCosmos ([124], [122]). It provides rich interactive parameterization capabilities for the generation and exploration of thematic software cities. Each of the representation elements can be configured interactively. For authorship towers, for example, the authors to be visualized can interactively be switched on or off. For the interaction with and navigation in the 3D scene standard navigation techniques (zoom, pan, rotate), coordinated views, stereo projection, and the use of special input devices (e.g. 3D mice) is provided. Additionally, the tool allows stepping forwards and backwards through the development history to study a software systems evolution in detail.

The tool is build on top of the JMonkeyEngine[2] (JME), an open source graphics library written in Java. It proved to be well capable of handling the visualization of large real-world software systems containing more than 10.000 classes.

Röder ([152] and Kossack ([106]) integrated the CrocoCosmos tool into the Eclipse IDE which is a first initial step towards both a seamless integration of software city visualizations into the software development process and a roundtrip visualization of software systems as discussed in chapter 8.2.

[2]jmonkeyengine.com, last access 24.02.2012

6.5 Summary

Many different approaches for the analysis and visualization of several aspects of software systems like quality and team organization are described in the literature. The strength of software city visualizations in general is the uniform way of representing these aspects in thematic maps that are coherently derived from one common geographical base model. In contrast to the standard data tables and diagrams, this clearly supports the cognitive process for integrating formerly separated aspects consistently into more complex mental maps.

In this chapter, we discussed the construction of thematic software cities on the basis of the EVOSTREETS layout approach. Thematic software cities are created by a thematic mapping which maps software analysis data to visual properties of software city elements. We provided several thematic mappings which have not been discussed for software cities before (e.g. authorships) and illustrated the expressiveness and consistency benefits of the EvoStreets approach for several academic, industrial, and open source systems.

The particular strength of EVOSTREETS based software cities results from a higher consistency during software evolution and an increased expressiveness which both result from encoding system evolution in the software city structure. Both characteristics allow for supporting new application scenarios for program comprehension and reverse engineering, quality analysis and assessment, and monitoring program evolution. The former fields of application benefit from the increased expressiveness of the EVOSTREETS layouts, whereas the latter field is enabled due to the high consistency the EVOSTREETS approach offers.

Everything is related to everything else,
but near things are more related than
distant things.

First Law of Geography
Waldo Tobler

Chapter 7

Software City Landscapes

The software cities described in the previous chapters represent software systems on the software design level, i.e. they typically depict software elements like classes, class dependencies, and their composition into larger package structures. This design level is an abstraction of the source code, and still for many application scenarios and for many project stakeholders it is yet too fine-grained. Instead, coarse-grained software structures like components and the dependencies between them are essential for many application scenarios throughout the development and maintenance particularly of large software systems. We refer to this abstraction level as the software systems' architecture as already discussed in chapter 3.1.

In this chapter we discuss how the coarse-grained architecture and the fine-grained design of software systems can be represented simultaneously in one coherent software city based visualization. For this purpose, we broaden the underlying metaphor from cities to landscapes: Software cities no longer depict entire software systems but the components these systems consist of. A meaningful placement of software cities within a software city landscape additionally allows for representing e.g. the overall dependency structure between components. The structure of each city represents the corresponding component's internal structure on the design level. Combining both abstraction levels into one coherent visualization in such a way provides simultaneous views on the software architecture and the software design level and allows for a smooth transition between both levels. Concerning the visualization pipeline in figure 1.1, this chapter again aims at the construction of a spatial software model.

Several aspects regarding the construction of software city landscapes are discussed in this chapter. We first discuss possible mappings from software structures (architecture and design) to software city landscapes. The placement of software cities in the software city landscape contributes significantly to the overall expressiveness and effectiveness of software city landscapes. This aspect is discussed in the second

section which presents two placement strategies. Afterwards, we discuss several criteria for the depiction of architecture level dependencies in software city landscapes. Two new approaches are presented. In section 4, finally, the benefits of combining cities and landscapes into one coherent visualization are demonstrated for two software systems.

7.1 Mapping Software Structures to Landscapes

Software systems are organized by means of decomposition on both the software architecture level as well as the software design level: Components form component hierarchies whereas design level entities like classes are typically organized in packages, namespaces, and the like. In the general case, there is no one to one mapping between these hierarchies in such a way that each component corresponds to one particular design level element (like a package). Instead, mappings between the software architecture and the software design can become rather complex. In the remainder of this chapter we assume that the software model (stored in the SWC) essentially is a system decomposition tree which from top to bottom is composed of an arbitrarily deep component hierarchy. Each leaf of this component hierarchy contains the component's design level decomposition tree that typically consists of design level elements like packages or classes.

Software city landscapes are hierarchically structured as well. They consist of software cities that in turn consist of a hierarchical street system and buildings. For the construction of software city landscapes both hierarchies, the software model hierarchy and the hierarchy provided by the metaphor, must be mapped to each other in such a way that both the system architecture and the system design are represented. Several mappings are possible. For example, cities may represent top level components. Top level components typically consist of finer-grained components. Since the design level must be represented, the cities in this case would have to depict a component hierarchy as well as the design hierarchy of each leaf component. In the context of this thesis, we decided for another mapping: Cities represent the finest-grained components of the system architecture, i.e. components that do not contain any subcomponents. In this case, cities have to represent the design of each component, only, i.e. they contain design level elements like classes and packages as done in the previous chapters. As a consequence, the software city landscapes discussed in the sequel include only a fraction of the architectural model (namely the set of finest-grained components, depicted as separate cities). They do not represent the component hierarchy defined in the architecture description.

7.2 City Placement

Two reasons motivate a discussion of positioning cities within landscapes. First, positions must have a clear meaning to avoid (possibly unconscious) misinterpretation. Second, placement can be used to represent additional data about the system structure which results in an increased expressiveness of the overall software landscape.

In this section, two placement strategies are discussed, i.e. placements depicting the intended system architecture (typically manually defined by some authority like the system's architect), and placements depicting the actual system architecture (typically extracted from the codebase). Both strategies were developed as part of a master thesis. They are described in [67].

7.2.1 Manual City Placement

As already pointed out in section 3.1.1, for many architectural patterns and reference architectures placement conventions regarding their visual representation have long been established. Tiers are usually drawn top-down with relations pointing from the top to the bottom tier whereas pipes and filters are usually arranged horizontally in alternating order with data flow pointing typically from the left to the right. Besides these commonly established guidelines, project specific placement conventions may implicitly be documented in respective architectural diagrams; GUI components for example are often drawn on top of other components whereas middleware and data storage components are often placed at the bottom of the diagram. In either case, to avoid confusion and misinterpretation these established placement conventions (tiers above each other, pipes and filters alternating from the left to the right, established reference architectures) should be taken into account when placing software cities in software landscapes.

The generic metamodel of the SOFTWARE COCKPIT allows for defining arbitrary architectural units like tiers and the like. Once these types are assigned to particular architectural units an automatic layout algorithm can determine an overall architectural layout that respects a set of placement conventions. Also, positions and schematic constraints can be defined and stored in the SWC so they can easily be regarded during landscape construction. This is a rather straight-forward topic, thus we do not further elaborate this strategy.

7.2.2 Force-Directed City Placement

Architectural diagrams are often guided by the principle that connected components should be placed in close proximity whereas disconnected components should be placed at larger distances. In case the architectural diagram and the actually implemented architecture in the system are compliant with each other, laying out software city landscapes according to the diagram would yield expressive and effective layouts since strongly connected components are depicted as cities with close proximity which additionally alleviates the visualization and interpretation of component dependencies. If the architecture documented in the architecture diagram and the actual system architecture are not compliant, then the overall landscape layout may be misinterpreted.

In chapter 2 several approaches for depicting relational data as two-dimensional maps were described, e.g. the vocabulary based thematic maps in [110] as well as call-graph and co-change graph based maps in [27]. These ideas can also be used for laying out software city landscapes to place cities in the landscape in such a way

that their distances reveal any of these relations. The results are highly expressive software city landscapes that support additional application scenarios, e.g. the analysis and monitoring of the actual component dependencies during evolution. In the sequel we discuss the use of force directed methods to construct software city landscapes. Because these landscapes represent relational data between software cities and to distinguish them from the aforementioned landscapes that are constructed by positioning cities manually, we refer to these landscapes as *relational software city landscapes*, or simply *relational landscapes*.

Force-directed algorithms are often used for graph visualization (e.g. [74], [143], [28], [27], [125]). They are based on the idea of modeling relational systems as systems of nodes and forces. Layouts are computed by systematically re-arranging the nodes with respect to the actually acting forces in the model such that the overall system energy is reduced. Two types of forces are typically used: Repulsing forces act on each pair of nodes of the graph and cause them to drift apart. Attracting forces act on nodes which are connected by a graph edge and cause them to move towards each other. Repulsing forces decrease in strength as two nodes drift apart and increase in strength as they converge; attracting forces show the opposite behavior. An optimization algorithm typically iterates repeatedly over the graph node set, computes all forces acting on each node, and moves nodes accordingly towards positions with lower remaining energy. The following discussions concentrate on particular aspects that have to be considered for placing software cities within landscapes. The construction of these landscapes requires a dependency graph, a force model, an optimization algorithm, and, in our particular case, a special distance measure for software cities (discussed below).

The Dependency Graph

The dependency graph stores the dependency structure that is to be laid out. It can be constructed in different ways: First, a *fine-grained* dependency graph contains nodes for each building in the city, and the edges of the graph represent the dependencies between the corresponding design level software entities. Second, a *coarse-grained* dependency graph contains one node for each city in the landscape, and the edges of the graph are aggregated edges representing the dependencies between the corresponding architecture level components. An additional edge weighting is used to indicate the strength of the aggregated edge. Clearly, the fine-grained dependency graph is significantly larger than the coarse-grained graph. Everth ([67]) analyzed both approaches and showed that the use of fine-grained dependency graphs requires significantly more computation time but, on the other hand, does not increase the quality of the visualization. Thus, we use the coarse grained approach in the remainder of this chapter.

Acting Forces: Attraction, Repulsion, Gravitation

In addition to attracting and repulsing forces between graph nodes we introduce another attracting force which allows for smoothly turning an architecture diagram based landscape into a relational landscape. An additional gravitation force attracts

each city to a manually defined position in the landscape. In other words, the force acts as an individual anchor for each software city. The anchor itself cannot be moved. It is defined on the basis of the manual placement strategy described above, i.e. with respect to a project's architectural diagram or on the basis of a placement convention.

Attraction, repulsion, and gravitation can globally be adjusted by linear factors which enables users to gradually turn a manually defined software landscape into a relational one, and vice versa.

Distance Measurement

In previous work, graph nodes often do not have an inner structure, nor do they have a particular shape and thus are considered being non-dimensional points; sometimes an additional node weight is used. The particular characteristic that needs to be considered here is that software cities are complex spatial objects. Thus, special care must be taken to avoid city intersection and to maintain the effectiveness of the software city landscapes. Force directed computations rely on a distance between each two nodes. As long as these nodes have no dimension their distance corresponds solely to the distance between their positions. Software cities, however, are composed of more complex two-dimensional shapes like buildings, streets, and possibly elevation landscapes. To avoid an intersection of these city elements we cannot just determine the smallest distance between these elements' center points. Instead, distances must be computed taking the particular shapes into account.

To reduce the complexity of this computation we use a simplification. The distance between each two cities is defined on the basis of the smallest enclosing circles, rectangles, or polygons which were already discussed in chapter 5. Regarding enclosing circles, distance is defined as the distance between their center points minus the sum of both radii. In terms of this definition, the distance between two cities is zero if their smallest enclosing circles intersect in one point.

Each of these shapes is a coarse approximation of the actual software city position and shape. As such, these approximations cause offsets to actual minimum distance between each two nodes of both cities. While the smallest enclosing convex polygon provides the best approximation of the actual shape the smallest enclosing circle is a rather coarse city description and thus could yield less accurate placements. However, the larger the actual distance between each two cities, the less influential this erroneous offset is. In the remainder of this chapter we use the smallest enclosing circle.

7.3 Edge Layouts

In software cities dependencies are typically depicted by straight lines or by curves constructed using additional edge bundling techniques to increase readability. These approaches work best in cases where a user can explore the visualization interactively and analyze it from different perspectives. For 2D projections, however,

these approaches often cause occlusion and intersections of uninvolved city elements which both cause a higher risk of misinterpretation. Within cities this occlusion can be resolved by stringent routing techniques like the street-routing approach where edges follow paths on the hierarchical street system, only. In software landscapes no such street infrastructure or similarly usable constructs are available, thus another solution must be found to effectively layout dependencies.

In this section, two new edge layout approaches for software landscapes are presented. Both were designed to address the following requirements:

- Depict component dependencies.
  Coarse component dependencies are of particular importance when analyzing the system on the architectural level, e.g. to test the system for architectural compliance. Thus, they should be easily cognizable.

- Preserve the expressiveness of the landscape layout.
  The layout of the landscape represents data about the dependency structure of the underlying system. Modifying this layout by re-positioning cities may distort this meaning and thus cause misinterpretation. Preserving the expressiveness of the layout thus implies that the layout must not change (in contrast to e.g. [61]).

- Avoid city intersections.
  City intersections cause occlusion and increase the risk of misinterpretation because the source and/or the target of individual edges may not be identifiable doubtlessly. Avoiding intersections thus preserves the visualization effectiveness even though other criteria (like short edge lengths) may be violated.

- Smooth transition from coarse-grained to fine-grained dependencies.
  While individual design level dependencies are less important when analyzing the component structure, they can be helpful for understanding and reasoning about the causes of architecture violations. Thus, a smooth transition from coarse-grained to fine-grained dependencies should be supported.

In the sequel, we will outline two ideas for laying out edges in software landscapes. The first approach was originally developed for the visualization of hyperedges in fixed graph layouts ([96]). It adopts the idea of force-directed landscape layouts. The second approach was developed in the context of a bachelor thesis ([182]). It uses a simple geometric layout algorithm. Both approaches were designed to preserve the expressiveness and effectiveness of the overall landscape layout as well as the individual city layouts. Preserving expressiveness here means that information must not be modified; preserving effectiveness means the visualization should not be capable of being misunderstood and that information must not be occluded. We present a new force-directed approach first and compare it with the new geometric approach afterwards.

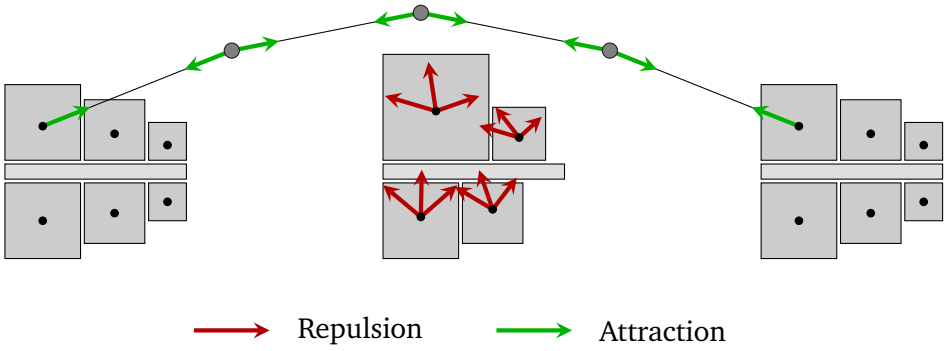


Figure 7.1: Force-Directed Edge Routing

7.3.1 Force-Directed Edge Layout

The idea behind this approach is that cities and edges repulse each other. The closer an edge circumvents a city, the stronger are the repulsing forces acting on the curve. A simple optimization algorithm then shifts the edge apart until a reasonably large distance is reached. Cities must not be shifted as this would disrupt the expressiveness of the overall landscape. To avoid infinite edge elongation we need an additional attracting force. Forces and the underlying graph model are discussed in the sequel.

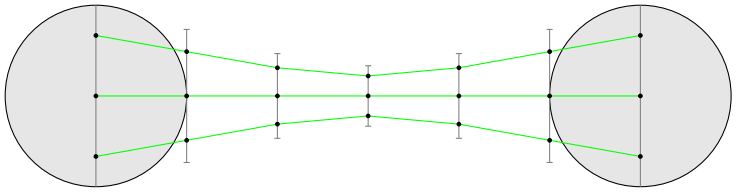
Curves and acting Forces

Figure 7.1 illustrates the approach. There are two types of nodes: Graph nodes (depicted in black) represent city elements (buildings) that must not be intersected and that must not be moved; their positions are fixed to preserve the landscape expressiveness. Curve nodes (depicted in light gray) represent the curve of the corresponding edge; they describe the route an edge actually follows. Thus, the graph that the minimization algorithm operates on is a bipartite graph that contains curve nodes for all edges and nodes for every building in the landscape.

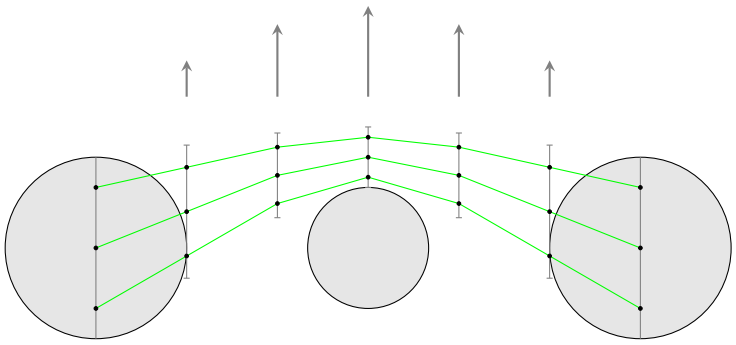
Repulsing forces act on curve nodes only, more precisely they cause translations for curve nodes only. The strength of these forces on each particular curve point depends on its position and distance to the graph nodes. To avoid an infinite elongation of curves attracting forces act between each two directly connected curve points. Attracting forces shorten a curve. They increase as the curve elongates, and decrease again as the curve is shortened. Repulsing forces show the opposite behavior.

Optimization Strategies

To compute a layout for a given edge, an optimization algorithm typically iterates over the curves node set (except start and end nodes which also have fixed positions), computes all forces acting on each node, and move each node towards positions with lower energy. Clearly, start and end point of each curve must not be



(a) Band Constriction



(a) Band Routing

Figure 7.2: Geometric Edge Routing

moved. The optimization algorithms may iterate over the node set from the beginning of a curve to its end, it could move the middle point at each curve first before recursively proceeding in the same manner with the left and right part of the curve, or it could move nodes according to the strength of all forces acting on it such that nodes which are placed worst are moved first.

Increasing the curve resolution, i.e. the number of curve nodes of each curve, increases both the quality of curve layouts as well as the necessary computation effort. Several strategies could reduce this effect, e.g. the use of force approximations as discussed in [17] or a simple filtering of graph nodes in close curve proximity. But depending on the curve resolution this computation still can become expensive rather quickly. To obtain an early approximation of curve layouts the algorithm used in this work uses a stepwise curve refinement. At the beginning, each curve consists of three curve points positioned at the start, center, and end of the curve. The center point now is moved according to the forces acting on it. Afterwards, the curve resolution is increased simply be splitting each curve segment into two smaller ones. The algorithm now performs node movements for each node again. Thus, the curve resolution is successively increased, and the computation may be cancelled by the user or if a predefined resolution is reached.

7.3.2 Geometric Edge Layout

An alternative approach to the force directed edge layouts is described by [182]. Figure 7.2 illustrates the idea. For each pair of software cities an elastic band is defined through which all dependencies between these two cities are led. By constricting the band (e.g. in a parabolic manner), edges between cities form cohesive bundles which are narrowest at the center between both cities and which broaden towards their source and target cities. The strength of constriction can be adjusted by user defined factors.

Constriction increases the visualization effectiveness since only edges connecting the same software cities are bundled. However, it does not avoid city intersection. For this purpose, the routes of bands are additionally adjusted. In case a band overlaps another city (which is approximated as e.g. smallest enclosing circle) the band (and thus all edges that are lead through it) is redirected by shifting it as illustrated in figure 7.2.

Depending on the resolution of the bands, constriction results in cohesive edge bundles between each two cities and band routing avoids city intersection.

7.3.3 Discussion

Uhlig ([182]) analyzed both approaches with respect to several quality criteria. In essence, his analysis revealed that force-directed edge routing does not avoid city intersections. Instead, edge routes may be trapped in local energy minima. Additionally, force-directed edge routing is computationally very expensive. The computation of the edge layouts depicted in figure 7.8 took about 77.6 seconds. In comparison, the geometric edge routing discussed above requires no more than a few (1-2) seconds even for high resolutions of the band.

7.4 Example Maps and Discussion

In this section, software landscapes for two software systems are discussed. The goal of this section is to illustrate the expressiveness and effectiveness of combining software landscapes and software cities in general and their particular benefits if these cities are built by the EVOSTREETS approach. For this purpose, we will discuss example visualizations of two software systems. The first is a rather small system, whose architecture and code mapping is explicitly documented. Second, we use the CrocoCosmos system and include several related projects.

7.4.1 Apache Shiro

Apache Shiro[1] is a security framework enabling authentication, authorization, cryptography, and session management. The systems architecture is well documented, also an explicit mapping from architecture to the code is available. Shiro is written in JAVA. We analyze project release `1.1.0` which contains 260 classes.

[1] http://shiro.apache.org, last access 01.03.2012

Component	Pattern
SECURITY MANAGER	`org.apache.shiro.mgt`
AUTHORIZER	`org.apache.shiro.authz`
AUTHENTICATOR	`org.apache.shiro.authc`
REALMS	`org.apache.shiro.realm`
SESSION	`org.apache.shiro.session`
CACHE	`org.apache.shiro.cache`
CRYPTOGRAPHY	`org.apache.shiro.crypto`

Table 7.1: Apache Shiro Component Mapping

Component Mapping

The project consists of seven explicitly defined components. Each component is implemented in a distinct JAVA package. A mapping from component to code package is listed in table 7.1. Altogether, these seven components comprise 174 classes. The remaining 86 classes are not assigned to a particular component and thus are not considered here. Figure[2] 7.3 shows the system's architecture as given on the project documentation.

Discussion

The goal of this discussion it to illustrate the benefits of software landscapes and edge routing mechanisms in general. The particular benefits of using these techniques in conjunction with EVOSTREETS based Software Cities are discussed for the CROCOCOSMOS project in the next section.

Figure 7.4 depicts a software city that shows the components of the Shiro system on the top level, i.e. each component is represented by a top level street branching off the main street. Components are depicted by unique colors which are maintained for all subsequent visualizations. Additionally, all dependencies between components are drawn, either as straight-lined edges or as routes computed by the force directed layout discussed above. The geometric edge routing is not applicable for such dense placements because the smallest enclosing circles overlap such that no intercity space for routing exists. Dependencies within components are omitted. The direction if each edge is encoded by a color from red to green saying that the entity depicted at the red end depends on the entity at the green end.

Obviously, components can easily be identified in the visualization, their dependency structure, however, remains widely unrecognizable, it is very difficult to extract from the visualization independent of whether the edges are laid out as straight-lined direct connections or as force-directed routes that at least reduce building intersection. Apart from a few exceptions, we cannot doubtlessly determine which components are related to each other and which are not, nor can the strength and dominant dependency direction between each two components be derived.

[2] http://shiro.apache.org/architecture.data/ShiroArchitecture.png, last access 01.03.2012

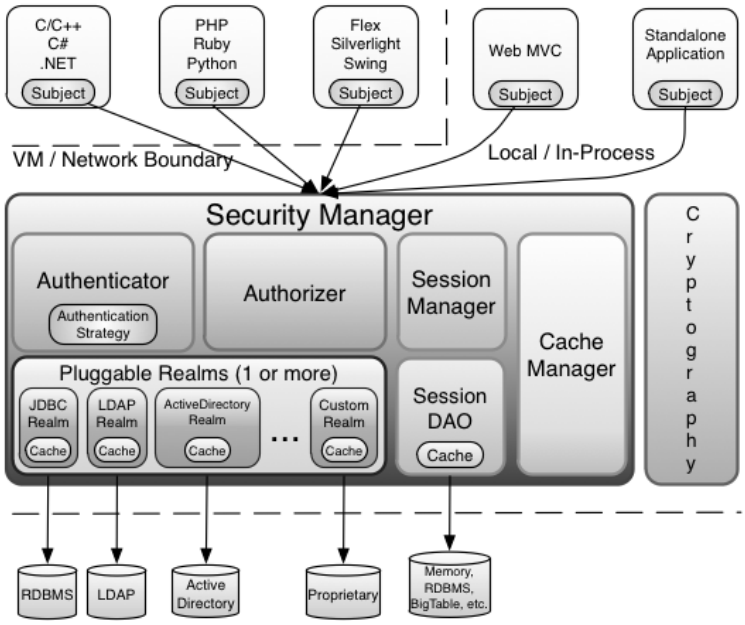
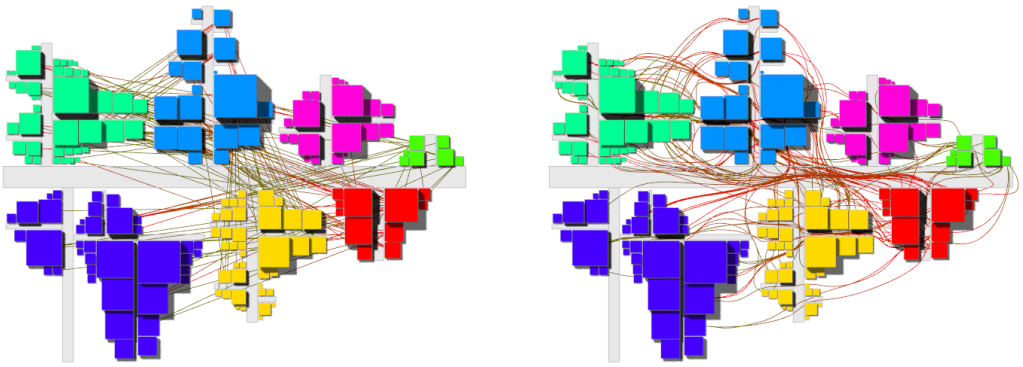


Figure 7.3: Apache Shiro Architecture



(a) Direct Edges (b) Force-Directed Edge Routing

Figure 7.4: Software City for Apache Shiro depicting Component Dependencies (174 classes)

Figure 7.5 depicts the same content as figure 7.4, however components are placed manually according to their arrangement depicted in the architectural diagram (figure 7.3): SECURITYMANAGER is placed above AUTHENTICATOR, AUTHORIZER, SESSION, and CACHE; it is horizontally centered with respect to these cities. SESSION and CACHE are positioned at the vertical center between the AUTHENTICATOR level and REALMS because these components span both layers in the architecture diagram. For a similar reason, CRYPTOGRAPHY is positioned on a lower level than SECURITYMANAGER, but still above SESSIONMANAGER and CACHE. The specific arrangement of this landscape is however of less importance in this context; clearly particular cities could easily be moved upwards or downwards to some degree without seriously impacting on the following discussion.

As above, only dependencies between components are shown but here they are laid out using the geometric edge routing approach. The overall coarse-grained component dependency structure now becomes clearly visible. This effect is mainly due to consistency of actual component dependency structure and the specified architecture diagram which of course is laid out in such a way that it expresses the intended dependencies. But in addition we also see that the geometric edge routing approach avoids city some intersections such as e.g. the dependency between REALMS and CACHEMANGER whose edge now clearly circumvents CACHE City.

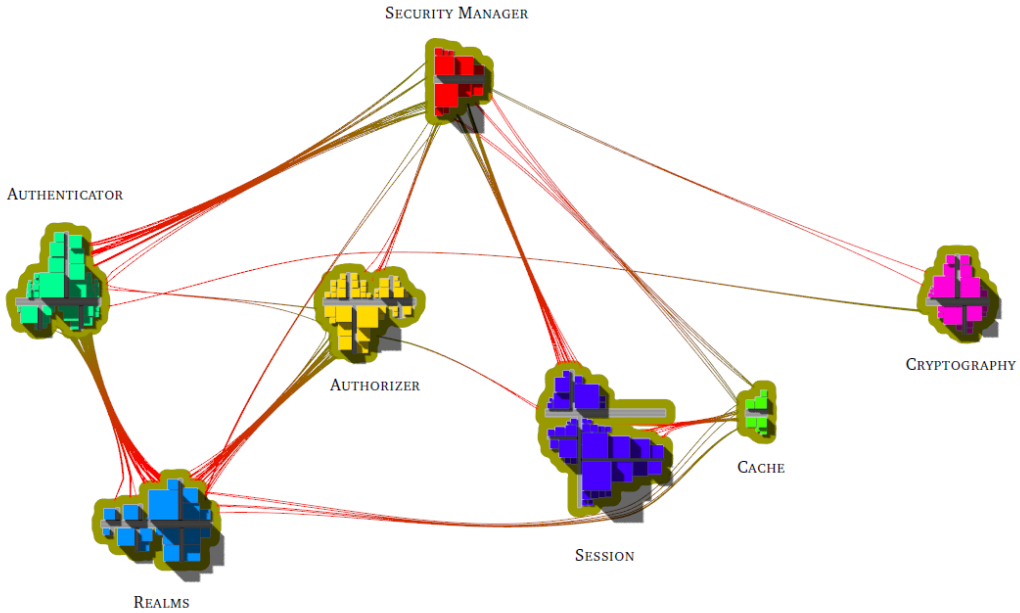


Figure 7.5: Software City Landscape for Apache Shiro, Manual City Placement (174 classes) with Geometric Edge Routing

In figure 7.6 the same content as in figure 7.5 is depicted. The landscape layout

this time is not specified manually but computed automatically by a force directed algorithm in such a way that strongly related components by trend are placed in close proximity to each other whereas loosely coupled components are placed at far distances. The algorithm started on a random grid based city placement and still it yields a layout with high similarity to the architectural diagram.

Another effect of this placement is the reduced number of edge intersections and the interpretability of city distances. Due to its relatively low coupling to other components CRYPTOGRAPHY City is placed at a relatively far distance to the remaining cities. In contrast, AUTHORIZER and REALMS are strongly connected indicated by the small inter-city distance.

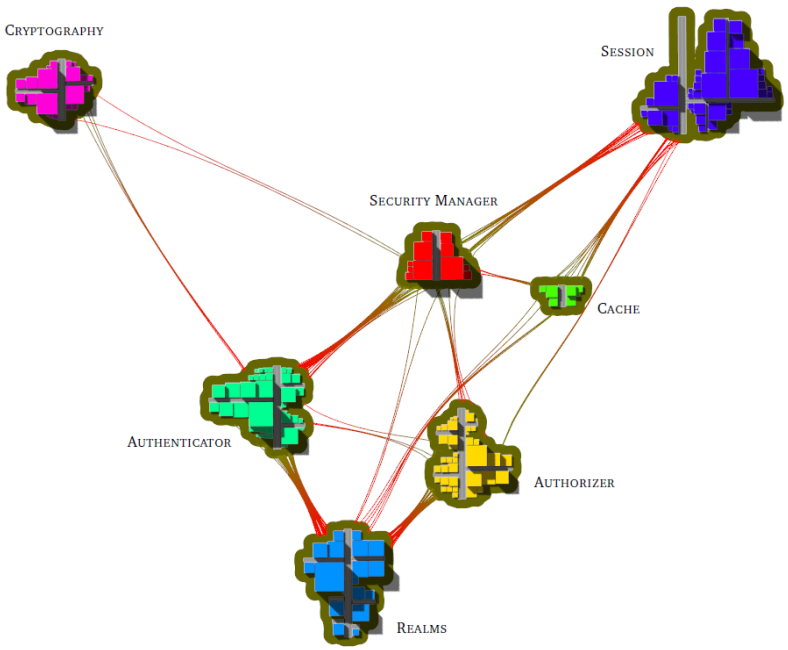


Figure 7.6: Software City Landscape for Apache Shiro, Force-Directed City Placement (174 classes) with Geometric Edge Routing

This section showed that for Apache Shiro a Software Landscape based on software cities yields more expressive and effective visualizations: First, dependencies on coarse component levels are much easier to read in comparison to if they are shown in cities only; second, even if omitted in the form of concrete visual representations, relational layout allows for conclusions on the overall dependency structure. On the other hand, the visualization obviously requires much more layout space and may also suffer from some degree of instability in case the underlying dependency structure changes.

7.4.2 CrocoCosmos

The goal of this section is to illustrate the particular benefits of using EVOSTREETS based software cities for software landscape visualization. Special focus is put on the expressiveness and effectiveness of the landscapes. We do not analyze the consistency because any such discussion returns the same results for every known software city construction discussed in the literature.

We choose CrocoCosmos again but broaden the project scope and include further projects which enhance the core project with specific functionality. Also, we use another temporal resolution and analyze 21 revisions uniformly distributed from revision `0` to revision `1000`.

Architecture and Component Mapping

CrocoCosmos involves five development projects with the following responsibilities:

- CROCOCLIPSE is a small component providing the necessary infrastructure to run CrocoCosmos and all its components as Rich Client Platform (RCP) application within the ECLIPSE environment.
- CROCOCOSMOS is the main component. It implements the second step of our visualization pipeline, i.e. the spatialization of software systems as treemaps, rectangle packings, and force directed relational layouts on which different software cities can be built. It provides the central data structures for all other components, offers a set of rudimentary 2D visualizations, and several analyses of layout quality.
- EVOVIEWER is the main component for the final visualization and interactive exploration of 3D software cities and landscapes. It implements the thematic mapping step of our visualization pipeline.
- SCENARIOVIEWER is a lightweight 3D viewer with strongly reduced capabilities that allows for exploring special purpose pre-defined visualizations in 3D. It is particularly useful for users exploring scenes that were previously exported from the EVOVIEWER.
- JME3FRAMEWORK is a service component that encapsulates common functionality that potentially is used by all 3D viewers. It provides central features like support for special input devices (3D mice), stereo rendering, image and scene export and more.

All these components are implemented in separate Java projects and follow a particular package naming convention: Each project is named using a common package prefix `org.btu.sst` followed by the respective component name. Table 7.2 gives an overview of all components and the mapping of these components to source code packages.
The EVOVIEWER and the SCENARIOVIEWER components both originate from the CROCOCOSMOS codebase. They evolved into separate components at some later

Component	Pattern
CROCOCLIPSE	`org.btu.sst.crococlipse`
CROCOCOSMOS	`org.btu.sst.crococosmo`
EVOVIEWER	`org.btu.sst.crococosmo.view.evoviewer`
	`org.btu.sst.crococosmo.view.evolutionViewer`
SCENARIOVIEWER	`org.btu.sst.crococosmo.view.scenarioviewer`
JME3FRAMEWORK	`org.btu.sst.jme3framework`

Table 7.2: CrocoCosmos Component Mapping

point in time. Thus, there are packages in the historic CROCOCOSMOS codebase which now belong to the EVOVIEWER or SCENARIOVIEWER components. Unless explicitly assigned otherwise, these code structures would actually appear in the CROCOCOSMOS city visualization in the form of Disused Sites. For all versions being visualized, we explicitly regard all content in the corresponding packages content of the EVOVIEWER or SCENARIOVIEWER, respectively, even if the actual implementation path has (for some period of time) been part of the CROCOCOSMOS project directory. Thus, CROCOCOSMOS includes all content below `org.btu.sst.crococosmo` except those packages listed for the viewer components.

Discussion

The layout for CROCOCOSMOS component differs from the layouts for CrocoCosmos system shown in previous chapters because of the fragmentation into several projects and a different temporal resolution. Figure 7.7 is a highly expressive visualization as it depicts the project's component structure including their dependencies, their internal decompositions, and their individual evolution. The landscape consists of two large cities (CrocoCosmos and EvoViewer), two smaller ones (ScenarioViewer and JME3Framework) and a rather small city (CrocoClipse). Beyond that, the following observations can be made:

- The city representations for CROCOCLIPSE and the SCENARIOVIEWER are uniformly surrounded by a terrain. Particularly, we cannot identify suburbs or mountain slopes. None of the city structures is placed on low elevation levels.
- The city representations for EVOVIEWER and CROCOCOSMOS provide all visual patterns discussed in chapter 4. Especially, suburbs and mountain slopes can be identified for both cities.
- CROCOCLIPSE is placed to the left of CROCOCOSMOS at far distance to all the other components. Further, there is a relatively large distance between CROCOCOSMOS and JME3FRAMEWORK, as well as between SCENARIOVIEWER and EVOVIEWER.

These three points tell a lot about the project. First, CROCOCLIPSE and SCENARIOVIEWER are old components which for a long period of time were not expanded anymore. This can be inferred from the visualization because of the thick uniform

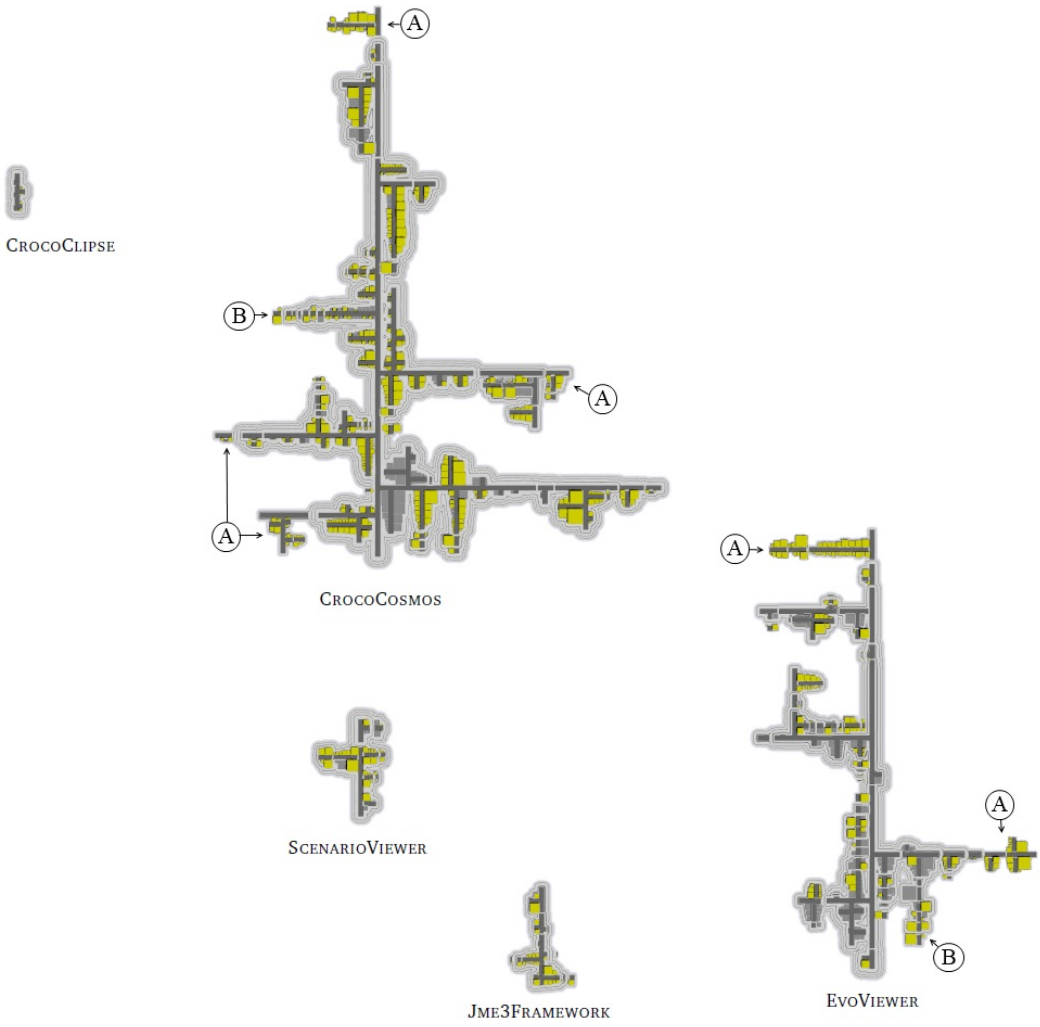


Figure 7.7: Software City Landscape for CrocoCosmos depicting component dependencies by proximity, component structure and evolution (789 classes)

terrain surrounding both cities. Given this observation, one might assume that these components are not developed any longer. As this is our own project we already know that the development for both has come to an end, there is no more development activity on these components. Project newcomers, however, could easily test this assumption by simply consulting a corresponding modification history map (as e.g. depicted in figure 6.8).

Second, the development seems to concentrate on the components CROCOCOSMOS and EVOVIEWER. For both components we can identify many structures forming suburbs (A) and mountain slopes (B), respectively. Especially the suburbs are reasonably large districts on rather low elevation levels which indicates that a large amount of new functionality is still being added to the system.

Third, concerning the arrangement of all five cities in the landscape, we can assume that CROCOCLIPSE depends on CROCOCOSMOS (or vice versa) because it is positioned to the left of CROCOCOSMOS and at a rather far distance to all other components. Also, the layout supports the interpretation of a relatively few couplings between ScenarioViewer and EvoViewer, as well as between CROCOCOSMOS and JME3FRAMEWORK. These interpretations can easily be confirmed by depicting the actual dependencies between all components as shown in figure 7.8. Both, SCENARIOVIEWER and EVOVIEWER depend on CROCOCOSMOS and the JME3FRAMEWORK, but they do not access each other, neither do CROCOCOSMOS and JME3FRAMEWORK. The CROCOCLIPSE component depends on the CROCOCOSMOS component.

Additionally, we now see that some of the recent system extensions increased the coupling particularly between the components CROCOCOSMOS and EVOVIEWER. The EVOVIEWER suburbs S1 and S2 strongly depend on the central data structure that is implemented in the CROCOCOSMOS component (the corresponding city region is annotated by DS). S1 additionally makes use of the CROCOCOSMOS suburb S3. In the following we discuss the reasons for these couplings.

S2 implements the feature we currently discuss, i.e. the positioning of software cities in landscapes. Like all layout approaches, S2 strongly accesses the central data structure of CROCOCOSMOS. S2, however, is not implemented in the CROCOCOSMOS component like most of the other layout computations. There are two reasons for this situation. First, a few layout approaches are realized in the EVOVIEWER component. The landscape computation requires access to each layout computation. Thus, adding it to the CROCOCOSMOS component would cause circular dependency between both components. Second, the landscape computation had to be added quickly to the system. To avoid a large restructuring of the EvoViewer component by moving all layout computations to the CROCOCOSMOS COMPONENT, we decided to add this feature to the EVOVIEWER component even though this had a negative impact on the cohesion of both components. Clearly, all layout computations should be moved to the CROCOCOSMOS component.

There is another interesting coupling between S1 and S3. Both suburbs were implemented by the same author (which is visible in the authorship map of this landscape) during the same period of time. They implement functions for visualizing the exe-

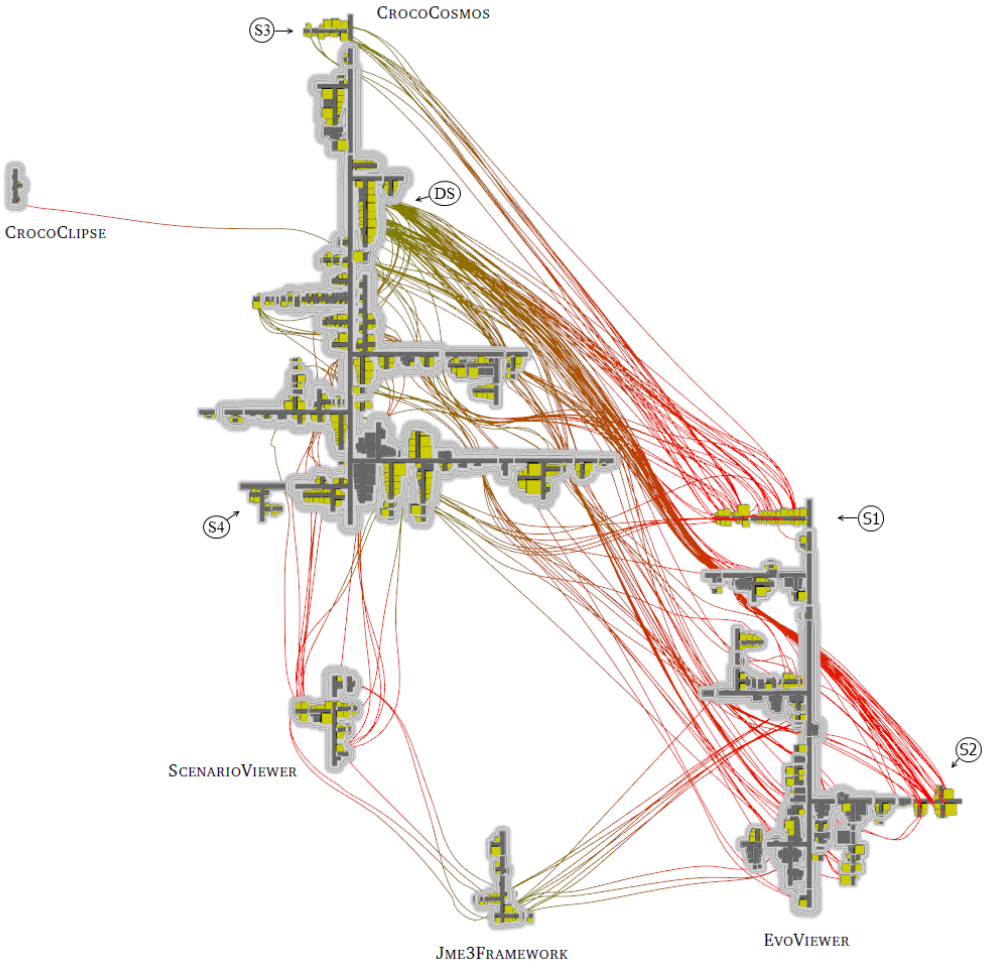


Figure 7.8: Software City Landscape for CrocoCosmos depicting all Dependencies between Components, Force-Directed Edge Layout (789 classes, computation time 77673 milliseconds)

cution of a program at runtime in software cities. S3 implements all functions that are necessary to record and process runtime data like memory usage or execution time. These data are stored into the central data structure DS. DS is accessed by S1. The coupling between S1 and S3 arises because S1 provides a user interface (in the EVOVIEWER component) to control the collection of runtime data (e.g. to start and stop the monitoring process, or to reset collected data).

An opposite example to S1, S2, and S3 is the CROCOCOSMOS suburb S4 which represents the layout consistency analysis we discussed in section 5. This computation does not use any other component since all the necessary data (node positions and node dimensions for each layout approach) is stored in the central graph (which belongs to the same component CROCOCOSMOS).

The dependencies depicted in figure 7.8 are laid out by the force directed edge routing approach discussed above. Bundles occur indirectly as the result of avoiding node intersections. However, bundles and edges may intersect cities at regions with low density, i.e. they are trapped in local minima. An example of such situation are the edges connecting S1 and DS in figure 7.8 which intersect the EVOVIEWER city twice instead of circumventing it (e.g. at the right hand side of S1).

7.5 Summary

Particularly for large software systems the coarse-grained software architecture is more relevant for many project stakeholders than the fine-grained software design. Most software cities and city like approaches described in prior research do, however, represent software systems on the software design level; they typically depict software elements like classes and class dependencies. In this chapter we described an extension of software cities, i.e. software city landscapes, that allows for simultaneously representing coarse-grained architecture components and their fine-grained internal design in one coherent, software city based visualization. In software city landscapes, the components of a software system are represented by software cities which are arranged in a software city landscape.

Two strategies for placing software cities in software city landscapes were discussed. The first strategy, i.e. the manual city placement, allows for regarding common placement conventions (e.g. for architectural patterns or reference architectures) or for constructing landscapes that are compliant with project specific architecture diagrams. The second strategy, i.e. the force-directed city placement, allows for representing the actual coarse-grained component dependencies by means of distances between the corresponding cities in the landscape and thus increases the expressiveness of the overall visualization. We also discussed two approaches for embedding representations of coarse-grained component dependencies into the landscapes.

Software city landscapes do not require the use of EVOSTREETS based software cities. Instead, all software city approaches proposed in the literature can be used. Also, different types of cities could be used in one software city landscape to represent different types of software components. For example, libraries or interface com-

ponents that do not change very often could be visualized using a highly compact representation as provided by the CodeCity approach, whereas components that are still under active development could be depicted as EVOSTREETS based cities.

A still open issue regarding software landscapes is that distances may only be interpreted between cities. Distances between software city elements, however, are strongly influenced by the internal structure of the respective software cities. Even strongly connected software elements may be represented by distantly positioned city elements. Tülling addressed this issue ([177]) and proposed several strategies to reduce the risk of misinterpreting distances between city elements including a rotation of cities, dynamically changing the order of city elements, or adjustable angles between streets. He analyzed the resulting layouts empirically by measuring the impact of these techniques on the lengths of all edges in the layout. The results show that rotating software cities actually reduces the overall edge lengths. However, Tülling also points out that in general the dependencies in software systems are rather complex and that the proposed techniques in general cannot avoid the problem of misinterpreting distances between software city elements.

Science may set limits to knowledge,
but should not set limits to imagination.

Bertrand Russell
(1872 - 1970)

Chapter 8

Summary and Outlook

8.1 Summary

Software visualization is a software engineering field which aims at providing visual representations of software systems or particular aspects thereof. Software cities denote one specific kind of software visualizations which adopts the metaphor of real cities. They typically depict the hierarchical software system decomposition by means of hierarchically nested city districts and buildings, i.e. logical containment relation is typically represented by a spatial containment relation of city elements. To support various comprehension and analysis scenarios additional software analysis data is often represented by visual properties of city elements. Previous research has shown the high expressiveness and effectiveness of software cities. However, they have not sufficiently addressed another important quality criterion, namely consistency. Consistency refers to the quality of visualizations to support continuous understanding of evolving data. It is particularly important when using software cities during ongoing software development and maintenance.

In this thesis, a new visualization approach that allows for a consistent visualization of evolving software systems as software cities during ongoing software development and maintenance is presented. It is based on a three-staged, fully tool supported visualization pipeline that allows for constructing expressive, effective, and consistent software cities. The first stage defines a *Software Model* that stores relevant software data and the evolution of software systems. The software model is populated by the Software Cockpit, a tool that can directly be integrated into standard build environments to continuously record the software structure and analysis data obtained from a wide range of analysis tools during its evolution. The second stage provides a *Spatial Software Model*. The spatial software model is derived from the software model by a processing step called *spatialization* that in essence maps inherently non-spatial data to a visualizable space. The purpose of the spatial software model is to provide an expressive, effective, and consistent city structure

that serves as a skeleton for anchoring further analysis data. The third stage, finally, adds thematic data (analysis data from a variety of software analysis tools) to the spatial software model. The resulting *Thematic Software Models* support specific comprehension, analysis, or monitoring scenarios.

The main contribution of this thesis is the EVOSTREETS layout approach which is derived from a systematic discussion of currently used layout approaches. The EVOSTREETS approach explicitly addresses layout consistency which despite its importance for real world software engineering has not appropriately been considered by prior research. The EVOSTREETS approach represents the structural evolution of software systems in the city structure in such a way that evolution becomes directly visible in the form of specific geographic patterns. Each of these patterns depicts particular evolutionary phenomena. Due to the higher expressiveness, EVOSTREETS based software cities allow for supporting new application scenarios in the field of program comprehension and reverse engineering as well as quality analysis and assessment. We evaluated several thematic software cities addressing these scenarios for a number of academic, industry, and open source software systems.

Another important effect of representing the software development history in the software city structure is that Software evolution does no longer disrupt the overall software city structure. Instead, software cities evolve smoothly during the system evolution. An empirical evaluation clearly revealed that EVOSTREETS layouts perform better with respect to layout consistency than those approaches that are used currently used for software cities.

We also described software city landscapes which allow for simultaneously representing coarse-grained software components and their dependencies as well as their fine-grained internal design in the one coherent, software city based visualization. In software city landscapes, components of software systems are represented by software cities. The placement of these cities in the software city landscape reveals coarse-grained component dependencies and thus increases the expressiveness of the overall approach. The depiction of software systems as software city landscapes is the first approach we are aware of that allows for simultaneously representing the hierarchical system decomposition from coarse-grained architecture components to fine-grained software design elements, the structural system evolution, and coarse-grained structural dependencies between components in one coherent visualization.

8.2 Extensions

- **Adoption of Cartographic Methods**
  The additional use of standard cartographic methods like height coloring, contour line annotations and the like may additionally increase the effectiveness of software cities. Particularly semantic zoom mechanisms would increase the readability of coarse-grained software structures, especially in software city landscapes. An initial semantic zoom approach for our software cities has recently been implemented in the context of a master thesis (cf. [19]).

Semantic zooming today is a standard means for exploring landscapes and city maps. Several map services like Google Maps[1], or OpenStreetMap[2] support semantic zooming and provide several further techniques for navigating maps, e.g. automatic labelling of map elements. Werner ([191]) analyzed the features these services provide and outlined a concept of how to adopt these map services to visualize software systems.

- **Increasing Layout Expressiveness for Analysis Scenarios**
  The selection of versions has a strong influence on the visual patterns that appear in the layout. Suburbs, for example, can resolve and turn into mountain slopes if the temporal resolution of the software model is increased. For monitoring scenarios the temporal resolution is defined by the monitoring process. For comprehension scenarios and retrospective analyses, however, the temporal resolution should be chosen carefully, e.g. with respect to particular development phases (like phases of strong growth). A best temporal resolution, maybe even for each street separately, clearly would increase the expressiveness of the visualizations and provide valuable insights into the development history.

- **Further Exploration of the Metaphor and the Layout Space**
  Panas et al. ([145]) already pointed out the high expressiveness of the city metaphor. Unfortunately, the ideas (the use of cars, clouds, fire, flashes, etc.) remained conceptual and have not been tested for real software systems. Adding these city elements clearly would increase the visualization expressiveness. Also, the layout space should be explored in further detail (e.g. the use of Voronoi treemaps ([13]) or other layout approaches).

- **User Study on Consistency**
  We analyzed the consistency of the EVOSTREETS approach using three similarity measures (that each addresses a specific similarity aspect) and three different analyses of the respective measurement results. A user experiment would be another way to characterize layout consistency.

- **Broader Coverage of the Software Life Cycle**
  Previous software city approaches mainly support comprehension and analysis scenarios in the context of reverse engineering or during maintenance. Our approach additionally supports scenarios during active development phases, particularly monitoring scenarios. However, all these approaches do not address early phases of software development. A city *plan* that represents the software architecture and design before active development begins (similar to the *UML City* discussed in chapter 2) would, for example, allow for better comparing development progress with project schedules.

Besides these extensions, one further aspect shall explicitly be stressed, i.e. the seamless integration of software visualization into the development process and the support for roundtrip visualization.

[1] maps.google.com, last access 27.03.2013
[2] www.openstreetmap.org, last access 27.03.2013

Seamless Integration and Roundtrip Visualization

Software Visualization has not reached software engineering, yet. There is a gap between software visualization research and industrial software engineering. In 1998, in the foreword of Stasko's et al. book on Software Visualization ([171]) Jim Foley states:

> "My only disappointment with the field is that Software Visualization has not yet had a major impact on the way we teach algorithms and programming or the ways in which we debug our programs and systems. While I continue to believe in the promise and potential of SV, it is at the same time the case that SV has not yet had the impact that many have predicted and hoped for."

More than 10 years later, the situation has not changed too much. Only few approaches (besides simple diagram representations) found their way into standard software engineering tools that are used in large, real world software projects. An adoption of elaborated visualization approaches like software cities by tool vendors and an integration of these approaches into standard software modeling and development tools like IDEs, VCSs and the like is still missing. Only very recently, a few analysis tools integrated the city metaphor for the visualization of software analysis data. For the time being, software visualization (unfortunately) is no widely used technology for software development.

Some authors cast doubt on software visualization in general (e.g. [36]), others discuss that software visualization has not yet addressed the right application scenarios (e.g. [153]). We believe that a seamless integration into development tools (an integration that does not cause any overhead) and the support for round-trip visualizations would significantly increase the interest in and acceptance of software visualization. While to some degree surely an engineering challenge, a seamless integration of software visualizations into development processes and tools is much more than only a technical requirement. It is an essential premise for acceptance by users. Software visualizations must not only be regarded as the result of some analysis chain, rather the visualization (more precisely the visual structures that are depicted) must become a first class engineering object. Besides the integration of software visualization into tools and processes, visualizations additionally must become round-trip ([44]) in such a way that the manipulation of a visual entity directly manipulates the corresponding software entity (as illustrated in e.g. [73]) and vice versa.

8.3 Beyond Software Visualization

EVOSTREETS based visualizations provide expressive, effective, and consistent depictions of many kinds of hierarchically structured, evolving data. Besides the visualization of software systems, it can be used for monitoring, analysis, or comprehension tasks in many other disciplines as well. For example, EVOSTREETS based

city representations allow for visualizing the evolution of a company's staffing or array of products, or the evolution of a university's student body. In the latter case, the university structure could be depicted as hierarchical street system representing e.g. faculties that consist of institutes which in turn organize particular courses of study. Students could be represented by *Student Towers* which are attached to the street that depicts the course of study of the respective student. Student towers could also depict student properties like gender, nationality and the like. Depending on the temporal resolution of the model the EVOSTREETS based city representation would form expressive geographic patterns indicating e.g. the beginning of each semester when new students matriculate, or courses of study that are aborted by many students within the first semesters. Also, disused sites with only very few remaining student towers would indicate long-term students and allow for analyzing their distribution among faculties, institutes, and so forth.

To illustrate that the approach is not restricted to software systems but works for other completely different domains as well we extracted data from an online collection of German laws and visualized these data. An example of the output is depicted in figure 8.1 which shows the current structure of the German BGB (Bürgerliches Gesetzbuch). In this city, paragraphs are depicted as towers. Their height represents the size of the corresponding paragraph (the number of clauses). The hierarchical organization of paragraphs into specific parts of the law (*Buch*, *Kapitel*, *Abschnitt* etc.) is represented by the hierarchical street system.

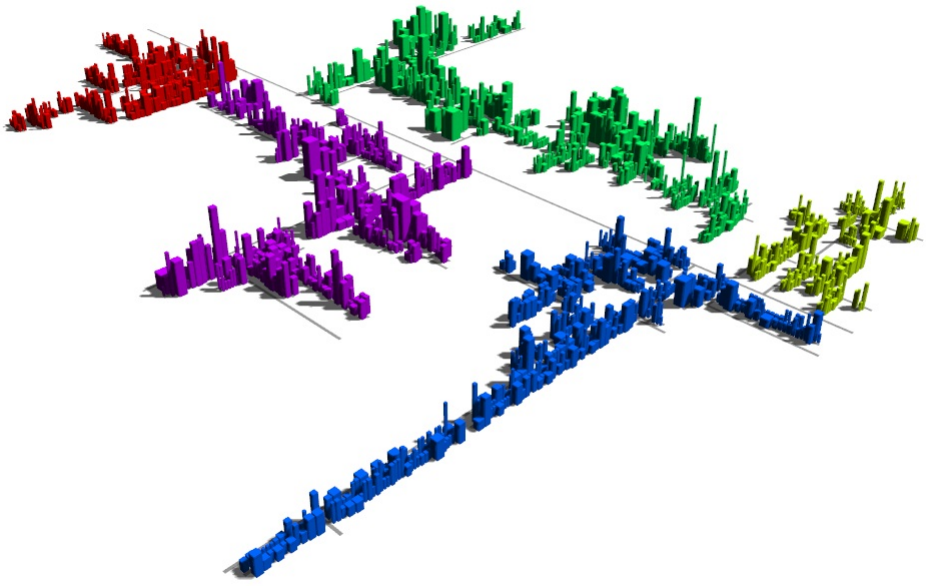


Figure 8.1: Depiction of the German BGB (Bürgerliches Gesetzbuch) as City

Clearly, there may be better means for discussing crime rates than, for example, the depiction of the number of convictions by paragraph in “conviction cities”. But this example demonstrates the applicability of our visualization pipeline for other domains.

Bibliography

[1] ABEL, Pierre ; GROS, Pascal ; SANTOS, Cristina Russo d. ; LOISEL, Didier ; PARIS, Jean-Pierre: Automatic construction of dynamic 3D metaphoric worlds: an application to network management. In: *SPIE Electronic Imaging 2000, 23-28 January 2000, San Jose, USA / Proceedings of SPIE – Volume 3960 Visual Data Exploration and Analysis VII, Robert F. Erbacher, Philip C. Chen, Jonathan C. Roberts, Craig M. Wittenbrink, Editors, February 2000*, 2000

[2] ALAM, Sazzadul ; DUGERDIL, Philippe: EvoSpaces: 3D Visualization of Software Architecture. In: *Proceedings 19th International Conference on Software Engineering and Knowledge Engineering (SEKE 2007)*, IEEE Computer Society, 2007, S. 500–505

[3] ALAM, Sazzadul ; DUGERDIL, Philippe: EvoSpaces Visualization Tool: Exploring Software Architecture in 3D. In: *Proceedings of the 14th Working Conference on Reverse Engineering (WCRE '07)*. Washington, DC, USA : IEEE Computer Society, 2007. – ISBN 0–7695–3034–6, S. 269–270

[4] ALPERT, Chuck J. ; NAM, Gi-Joon ; VILLARRIBUA, Paul ; YILDIZ, Mehmet C.: Placement stability metrics. In: *Proceedings of the 2005 Asia and South Pacific Design Automation Conference*. New York, NY, USA : ACM, 2005 (ASP-DAC '05). – ISBN 0–7803–8737–6, S. 1144–1147

[5] ANDREWS, Keith: Visual Exploration of Large Hierarchies with Information Pyramids. In: *Proceedings of International Conference on Information Visualisation*. Los Alamitos, CA, USA : IEEE Computer Society, 2002. – ISSN 1093–9547, S. 793

[6] ANTONIOL, Giuliano ; PENTA, Massimiliano D. ; MERLO, Ettore: An Automatic Approach to identify Class Evolution Discontinuities. In: *Proceedings of the Principles of Software Evolution, 7th International Workshop*. Washington, DC, USA : IEEE Computer Society, 2004. – ISBN 0–7695–2211–4, S. 31–40

[7] ANVIK, John ; MURPHY, Gail C.: Determining Implementation Expertise from Bug Reports. In: *Proceedings of the Fourth International Workshop on Mining Software Repositories (MSR '07)*. Washington, DC, USA : IEEE Computer Society, 2007. – ISBN 0–7695–2950–X, S. 2

[8] AVERBUKH, V. L.: Visualization Metaphors. In: *Programming and Computer Software* 27 (2001), September, S. 227–237. – ISSN 0361–7688

[9] BAECKER, Ronald ; PRICE, Blaine: The Early History of Software Visualization. In: STASKO, John (Hrsg.) ; DOMINGUE, John (Hrsg.) ; BROWN, Marc H. (Hrsg.) ; PRICE, Blaine A. (Hrsg.): *Software Visualization, Programming as a Multimedia experience*, The MIT Press, 1997. – ISBN 0–262–19395–7

[10] BALL, Thomas ; EICK, Stephen G.: Software Visualization in the Large. In: *Computer* 29 (1996), S. 33–43. – ISSN 0018–9162

[11] BALL, Thomas ; KIM, Jung-Min ; PORTER, Adam A. ; SIY, Harvey P.: If your version control system could talk ... In: *Proceedings of the ICSE '97 Workshop on Process Modelling and Empirical Studies of Software Engineering*, 1997

[12] BALZER, Michael ; DEUSSEN, Oliver: Hierarchy Based 3D Visualization of Large Software Structures. In: *Proceedings of the conference on Visualization '04*. Washington, DC, USA : IEEE Computer Society, 2004 (VIS '04). – ISBN 0–7803–8788–0, S. 598

[13] BALZER, Michael ; DEUSSEN, Oliver: Voronoi Treemaps. In: *IEEE Symposium on Information Visualization* 0 (2005), S. 7. – ISSN 1522–404x

[14] BALZER, Michael ; DEUSSEN, Oliver: Level-of-detail visualization of clustered graph layouts. In: *Asia-Pacific Symposium on Visualization* 0 (2007), S. 133–140. ISBN 1–4244–0808–3

[15] BALZER, Michael ; DEUSSEN, Oliver ; LEWERENTZ, Claus: Voronoi treemaps for the visualization of software metrics. In: *Proceedings of the 2005 ACM symposium on Software visualization (SoftVis '05)*. New York, NY, USA : ACM, 2005. – ISBN 1–59593–073–6, S. 165–172

[16] BALZER, Michael ; NOACK, Andreas ; DEUSSEN, Oliver ; LEWERENTZ, Claus: Software Landscapes: Visualizing the Structure of Large Software Systems. In: *Proceedings of the Joint Eurographics - IEEE TCVG Symposium on Visualization (VisSym 2004)*. Konstanz, Germany : Eurographics Association, 2004, S. 261–266

[17] BARNES, Josh ; HUT, Piet: A hierarchical O(N log N) force-calculation algorithm. In: *Nature* 324 (1986), S. 446–449

[18] BARTELS, Max: *Evaluation von Stadt-Visualisierungen für Software-Systeme*. Bachelor Thesis, Brandenburg University of Technology, Cottbus, Germany, 2009

[19] BARTELS, Max: *Entwicklung eines Konzepts für den semantischen Zoom in der Ansicht einer Softwarestadt*, Brandenburg University of Technology, Cottbus, Germany, Diplomarbeit, 2012

[20] BASSIL, Sarita ; KELLER, Rudolf K. ; IRO, Département: Software visualization tools: Survey and analysis. In: *Proceedings of the 9 th International Workshop on Program Comprehension*, IEEE Computer Society Press, 2001, S. 7–17

[21] BAUR, Michael ; SCHANK, Thomas: Dynamic Graph Drawing in Visone / Fakultät für Informatik, Universität Karlsruhe. 2008-05. – Forschungsbericht

[22] BEDERSON, Benjamin B.: PhotoMesa: a zoomable image browser using quantum treemaps and bubblemaps. In: *Proceedings of the 14th annual ACM symposium on User interface software and technology*. New York, NY, USA : ACM, 2001 (UIST '01). – ISBN 1–58113–438–X, S. 71–80

[23] BEDERSON, Benjamin B. ; SHNEIDERMAN, Ben ; WATTENBERG, Martin: Ordered and quantum treemaps: Making effective use of 2D space to display hierarchies. In: *ACM Transactions on Graphics* 21 (2002), October, S. 833–854. – ISSN 0730–0301

[24] BENNICKE, Marcel ; STEINBRÜCKNER, Frank ; RADICKE, Mathias ; RICHTER, Jan-Peter: Das sd&m Software Cockpit: Architektur und Erfahrungen. In: KOSCHKE, Rainer (Hrsg.) ; HERZOG, Otthein (Hrsg.) ; RÖDIGER, Karl-Heinz (Hrsg.) ; RONTHALER, Marc (Hrsg.): *GI Jahrestagung (2)* Bd. 110, GI, 2007 (LNI). – ISBN 978–3–88579–204–8, S. 254–260

[25] BERTIN, Jacques: *Semiology of graphics*. University of Wisconsin Press, 1983. – ISBN 978–0–299–09060–9

[26] BEYER, Dirk ; HASSAN, Ahmed E.: Animated Visualization of Software History using Evolution Storyboards. In: *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE '06)*. Washington, DC, USA : IEEE Computer Society, 2006. – ISBN 0–7695–2719–1, S. 199–210

[27] BEYER, Dirk ; HASSAN, Ahmed E.: Evolution Storyboards: Visualization of Software Structure Dynamics. In: *Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC 2006, Athens, June 14-16)*, IEEE Computer Society Press, Los Alamitos (CA), 2006, S. 248–251

[28] BEYER, Dirk ; NOACK, Andreas: Clustering Software Artifacts Based on Frequent Common Changes. In: *Proceedings of the 13th International Workshop on Program Comprehension (IWPC '05)*. Washington, DC, USA : IEEE Computer Society, 2005. – ISBN 0–7695–2254–8, S. 259–268

[29] BLÄUL, Andre: *Constraints für Energie-basierte Graphlayouts*, Brandenburg University of Technology, Cottbus, Germany, Diplomarbeit, 2010

[30] BOCCUZZO, Sandro ; GALL, Harald C.: CocoViz: Towards Cognitive Software Visualizations. In: *Proceedings of IEEE International Workshop on Visualizing Software for Understanding and Analysis (VisSoft 2007)*, IEEE Computer Society, 2007

[31] BOHNET, Johannes ; DÖLLNER, Jürgen: Monitoring code quality and development activity by software maps. In: *Proceedings of the 2nd Workshop on Managing Technical Debt*. New York, NY, USA : ACM, 2011 (MTD '11). – ISBN 978–1–4503–0586–0, S. 9–16

[32] BRANDES, U. ; WILLHALM, T.: Visualization of bibliographic networks with a reshaped landscape metaphor. In: *Proceedings of the symposium on Data Visualisation 2002*. Aire-la-Ville, Switzerland, Switzerland : Eurographics Association, 2002 (VISSYM '02). – ISBN 1–58113–536–X, 159–ff

[33] BRANKE, Jürgen: Dynamic Graph Drawing. In: KAUFMANN, Michael (Hrsg.) ; WAGNER, Dorothea (Hrsg.): *Drawing Graphs* Bd. 2025. Springer Berlin / Heidelberg, 2001, S. 228–246

[34] BRIDGEMAN, Stina ; TAMASSIA, Roberto: Difference Metrics for Interactive Orthogonal Graph Drawing Algorithms. In: WHITESIDES, Sue (Hrsg.): *Graph Drawing* Bd. 1547. Springer Berlin / Heidelberg, 1998, S. 57–71

[35] BRIDGEMAN, Stina ; TAMASSIA, Roberto: A User Study in Similarity Measures for Graph Drawing. In: MARKS, Joe (Hrsg.): *Graph Drawing* Bd. 1984. Springer Berlin / Heidelberg, 2001, S. 231–235

[36] BROOKS, Frederick P. Jr.: No Silver Bullet Essence and Accidents of Software Engineering. In: *Computer* 20 (1987), April, S. 10–19. – ISSN 0018–9162

[37] BRULS, Mark ; HUIZING, Kees ; WIJK, Jarke van: Squarified Treemaps. In: *Proceedings of the Joint Eurographics and IEEE TCVG Symposium on Visualization*, Press, 1999, S. 33–42

[38] BYELAS, H. ; TELEA, A.: Visualization of areas of interest in software architecture diagrams. In: *Proceedings of the 2006 ACM symposium on Software Visualization (SoftVis)*. New York, NY, USA : ACM, 2006. – ISBN 1–59593–464–2, S. 105–114

[39] CARD, Stuart K. (Hrsg.) ; MACKINLAY, Jock D. (Hrsg.) ; SHNEIDERMAN, Ben (Hrsg.): *Readings in information visualization: using vision to think*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1999. – ISBN 1–55860–533–9

[40] CASERTA, Pierre ; ZENDRA, Olivier: Visualization of the Static Aspects of Software: A Survey. In: *IEEE Transactions on Visualization and Computer Graphics* 17 (2011), S. 913–933. – ISSN 1077–2626

[41] CASERTA, Pierre ; ZENDRA, Olivier ; BODENES, Damien: 3D Hierarchical Edge bundles to visualize relations in a software city metaphor. In: *Proceedings of International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*, IEEE, 2011. – ISBN 978–1–4577–0822–0, S. 1–8

[42] CHALMERS, Matthew: Using a landscape metaphor to represent a corpus of documents. In: FRANK, Andrew (Hrsg.) ; CAMPARI, Irene (Hrsg.): *Spatial Information Theory A Theoretical Basis for GIS* Bd. 716. Springer Berlin / Heidelberg, 1993, S. 377–390

[43] CHARTERS, Stuart M. ; KNIGHT, Claire ; THOMAS, Nigel ; MUNRO, Malcolm: Visualisation for informed decision making; from code to components. In: *SEKE '02: Proceedings of the 14th international conference on Software engineering and knowledge engineering*. New York, NY, USA : ACM, 2002. – ISBN 1–58113–556–4, S. 765–772

[44] CHARTERS, Stuart M. ; THOMAS, Nigel ; MUNRO, Malcolm: The end of the line for Software Visualisation? In: *Proceedings of 2nd IEEE Workshop on Visualizing Software For Analysis and Understanding (VISSOFT '03)*, 2003

[45] CHEN, Chaomei: *Information Visualisation and Virtual Environments*. Springer-Verlag London Limited, 1999. – ISBN 1–85233–136–4

[46] CHEN, Chaomei: *Mapping Scientific Frontiers: The Quest for Knowledge Visualization*. Springer-Verlag New York, Inc., 2003. – ISBN 1–85233–494–0

[47] CHEN, Chaomei ; PAUL, Ray J.: Visualizing a Knowledge Domain's Intellectual Structure. In: *Computer* 34 (2001), March, S. 65–71. – ISSN 0018–9162

[48] CHIDAMBER, S. R. ; KEMERER, C. F.: A Metrics Suite for Object Oriented Design. In: *IEEE Trans. Softw. Eng.* 20 (1994), June, S. 476–493. – ISSN 0098–5589

[49] CHIU, Patrick ; GIRGENSOHN, Andreas ; LERTSITHICHAI, Surapong ; POLAK, Wolf ; SHIPMAN, Frank: MediaMetro: browsing multimedia document collections with a 3D city metaphor. In: *MULTIMEDIA '05: Proceedings of the 13th annual ACM international conference on Multimedia*. New York, NY, USA : ACM, 2005. – ISBN 1–59593–044–2, S. 213–214

[50] CHRISTENSEN, Henrik B.: Utilising a Geographic Space Metaphor in a Software Development Environment. In: *Proceedings of the IFIP TC2/TC13 WG2.7/WG13.4 Seventh Working Conference on Engineering for Human-Computer Interaction*. Deventer, The Netherlands : Kluwer, B.V., 1999. – ISBN 0–412–83520–7, S. 39–56

[51] CHRISTENSEN, Henrik B. ; HANSEN, Klaus M.: Towards architectural information in implementation: NIER track. In: *Proceeding of the 33rd international conference on Software engineering*. New York, NY, USA : ACM, 2011 (ICSE '11). – ISBN 978–1–4503–0445–0, S. 928–931

[52] D'AMBROS, Marco ; LANZA, Michele ; LUNGU, Mircea: The evolution radar: visualizing integrated logical coupling information. In: *Proceedings of the 2006 international workshop on Mining software repositories*. New York, NY, USA : ACM, 2006 (MSR '06). – ISBN 1–59593–397–2, 26–32

[53] DELINE, Robert: Staying Oriented with Software Terrain Maps. In: *Workshop on Visual Languages and Computation*, 2005

[54] DeLine, Robert ; Czerwinski, Mary ; Meyers, Brian ; Venolia, Gina ; Drucker, Steven ; Robertson, George: Code Thumbnails: Using Spatial Memory to Navigate Source Code. In: *IEEE Symposium on Visual Languages and Human-Centric Computing* 0 (2006), S. 11–18. ISBN 0–7695–2586–5

[55] DeLine, Robert ; Venolia, Gina ; Rowan, Kael: Software Development with Code Maps. In: *Queue* 8, S. 10:10–10:18. – ISSN 1542–7730

[56] Dieberger, Andreas ; Frank, Andrew U.: A City Metaphor to Support Navigation in Complex Information Spaces. In: *Journal of Visual Languages and Computing* 9 (1998), Nr. 6, S. 597–622

[57] Diehl, Stephan: *Software Visualization - Visualizing the Structure, Behaviour, and Evolution of Software*. Springer, 2007. – ISBN 978–3–540–46504–1

[58] Diehl, Stephan ; Görg, Carsten: Graphs, They Are Changing. In: *Revised Papers from the 10th International Symposium on Graph Drawing*. London, UK : Springer-Verlag, 2002 (GD '02). – ISBN 3–540–00158–1, S. 23–30

[59] Dubuisson, M. P. ; Jain, A. K.: A modified Hausdorff distance for object matching. In: *Proceedings of 12th International Conference on Pattern Recognition* Bd. 1, IEEE Comput. Soc. Press, 1994. – ISBN 0–8186–6265–4, S. 566–568

[60] Dugerdil, Philippe ; Alam, Sazzadul: Execution Trace Visualization in a 3D Space. In: *Proceedings of the Fifth International Conference on Information Technology: New Generations (ITNG '08)*. Washington, DC, USA : IEEE Computer Society, 2008. – ISBN 978–0–7695–3099–4, S. 38–43

[61] Dwyer, Tim ; Marriott, Kim ; Wybrow, Michael: Integrating Edge Routing into Force-Directed Layout. In: Kaufmann, Michael (Hrsg.) ; Wagner, Dorothea (Hrsg.): *Graph Drawing* Bd. 4372. Springer Berlin / Heidelberg, 2007, S. 8–19

[62] Eades, Peter ; Lai, Wei ; Misua, Kazuo ; Sugiyama, Kozo: Preserving the mental map of a diagram. In: *Proceedings of COMPUGRAPHICS*, 1991, S. 24–33

[63] Earnshaw, R. A. ; Wiseman, Norman: *An introductory guide to scientific visualization*. New York, NY, USA : Springer-Verlag New York, Inc., 1992. – ISBN 0–387–54664–2

[64] Eick, Stephen G.: Graphically displaying text. In: *Journal of Computational and Graphical Statistics* 3 (1994), S. 127–142

[65] Eick, Stephen G. ; Graves, Todd L. ; Karr, Alan F. ; Mockus, Audris ; Schuster, Paul: Visualizing Software Changes. In: *IEEE Transactions on Software Engineering* 28 (2002), April, S. 396–412. – ISSN 0098–5589

[66] Eick, Stephen G. ; Steffen, Joseph L. ; Sumner, Eric E. Jr.: Seesoft - A Tool for Visualizing Line Oriented Software Statistics. In: *IEEE Transactions on Software Engineering* 18 (1992), S. 957–968. – ISSN 0098–5589

[67] Everth, Martin: *Visualizing Software Architectures in Software Cities*, Brandenburg University of Technology, Cottbus, Germany, Diplomarbeit, 2011

[68] Fabrikant, Sara I. ; Montello, Daniel R. ; Mark, David M.: The natural landscape metaphor in information visualization: The role of commonsense geomorphology. In: *J. Am. Soc. Inf. Sci. Technol.* 61 (2010), February, S. 253–270. – ISSN 1532–2882

[69] Fischer, Michael ; Pinzger, Martin ; Gall, Harald: Populating a Release History Database from Version Control and Bug Tracking Systems. In: *Proceedings of the International Conference on Software Maintenance*. Washington, DC, USA : IEEE Computer Society, 2003 (ICSM '03). – ISBN 0–7695–1905–9, S. 23 ff

[70] FOWLER, Martin: *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA : Addison-Wesley, 1999. – ISBN 0–201–48567–2

[71] FRISHMAN, Yaniv ; TAL, Ayellet: Online Dynamic Graph Drawing. In: *IEEE Transactions on Visualization and Computer Graphics* 14 (2008), Nr. 4, S. 727–740. – ISSN 1077–2626

[72] FROEHLICH, Jon ; DOURISH, Paul: Unifying Artifacts and Activities in a Visual Tool for Distributed Software Development Teams. In: *Proceedings of the 26th International Conference on Software Engineering*. Washington, DC, USA : IEEE Computer Society, 2004 (ICSE '04). – ISBN 0–7695–2163–0, S. 387–396

[73] FRONK, A. ; ALFERT, K.: Manipulation of 3-Dimensional Visualizations of Java Class Relations. In: *Proceedings of the 6th World Conference on Integrated Design Process Technology*, Society for Design and Process Science, 2002

[74] FRUCHTERMAN, Thomas M. J. ; REINGOLD, Edward M.: Graph drawing by force-directed placement. In: *Softw. Pract. Exper.* 21 (1991), November, Nr. 11, S. 1129–1164. – ISSN 0038–0644

[75] GALL, Harald ; HAJEK, Karin ; JAZAYERI, Mehdi: Detection of Logical Coupling Based on Product Release History. In: *Proceedings of the International Conference on Software Maintenance (ICSM 1998)*, IEEE Computer Society, 1998, S. 190–197

[76] GARLAN, David ; SHAW, Mary: An introduction to software architecture. In: AMBRIOLA, V. (Hrsg.) ; TORTORA, G. (Hrsg.): *Advances in Software Engineering and Knowledge Engineering* Bd. I, World Scientific Publishing Company, 1993, S. 1–39

[77] GIRBA, Tudor ; KUHN, Adrian ; SEEBERGER, Mauricio ; DUCASSE, Stephane: How Developers Drive Software Evolution. In: *International Workshop on Principles of Software Evolution* 0 (2005), S. 113–122. – ISSN 1550–4077

[78] GIRBA, Turdor: *Modeling History to Understand Software Evolution*, University of Berne, Switzerland, Diss., 2007

[79] GORTON, Ian: *Essential software architecture*. Springer, 2006. – ISBN 9783540287131

[80] GRACANIN, Denis ; MATKOVIC, Kresimir ; ELTOWEISSY, Mohamed: Software visualization. In: *Innovations in Systems and Software Engineering* 1 (2005), S. 221–230

[81] GRAHAM, Hamish ; YANG, Hong Y. ; BERRIGAN, Rebecca: A solar system metaphor for 3D visualisation of object oriented software metrics. In: *APVis '04: Proceedings of the 2004 Australasian symposium on Information Visualisation*. Darlinghurst, Australia : Australian Computer Society, Inc., 2004. – ISBN 1–920682–17–1, S. 53–59

[82] GRANITZER, Michael ; KIENREICH, Wolfgang ; SABOL, Vedran ; ANDREWS, Keith ; KLIEBER, Werner: Evaluating a System for Interactive Exploration of Large, Hierarchically Structured Document Repositories. In: *INFOVIS '04: Proceedings of the IEEE Symposium on Information Visualization*. Washington, DC, USA : IEEE Computer Society, 2004. – ISBN 0–7803–8779–3, S. 127–134

[83] GRISWOLD, William G. ; YUAN, Jimmy J. ; KATO, Yoshikiyo: Exploiting the map metaphor in a tool for software evolution. In: *Proceedings of the 23rd International Conference on Software Engineering*. Washington, DC, USA : IEEE Computer Society, 2001 (ICSE '01). – ISBN 0–7695–1050–7, S. 265–274

[84] HANNEMANN, Jan ; KICZALES, Gregor: Overcoming the Prevalent Decomposition of Legacy Code. In: *In Workshop on Advanced Separation of Concerns*, 2001

[85] HELM, Richard ; GAMMA, Erich ; VLISSIDES, John ; JOHNSON, Ralph: *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995 (Addison-Wesley professional computing series). – ISBN 9780201633610

[86] HESS, Andreas ; HUMM, Bernhard ; VOSS, Markus ; ENGELS, Gregor: Structuring Software Cities A Multidimensional Approach. In: *Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference*. Washington, DC, USA : IEEE Computer Society, 2007. – ISBN 0–7695–2891–0, 122 ff

[87] HESSELBARTH, Ulf: *Konzeption, Integration und Evaluation einer szenarienorientierten Visualisierung für das Capgemini sd&m Software-Cockpit*, Brandenburg University of Technology, Cottbus, Germany, Diplomarbeit, 2010

[88] HOLT, Ric: Software Architecture as a Shared Mental Model. In: *ASERC Workshop on Software Architecture*, University of Alberta, 2001

[89] HOLZ, Philipp: *Analyse und Optimierung der Stabilität und Kompaktheit zweier Layoutverfahren für Softwarestädte*. Bachelor Thesis, Brandenburg University of Technology, Cottbus, Germany, 2010

[90] HTTP://WWW.SEI.CMU.EDU/ARCHITECTURE/START/COMMUNITY.CFM: *Software Architecture Community Definitions*. Online Collection, accessed 08/2011

[91] IEEE: Recommended practice for architectural description of software-intensive systems. In: *IEEE Std 1471-2000* (2000), S. i–23

[92] JOHNSON, Brian ; SHNEIDERMAN, Ben: Tree-Maps: a space-filling approach to the visualization of hierarchical information structures. In: *Proceedings of the 2nd conference on Visualization '91*. Los Alamitos, CA, USA : IEEE Computer Society Press, 1991 (VIS '91). – ISBN 0–8186–2245–8, S. 284–291

[93] JOHNSON, Mark: *The Body in the Mind*. The University of Chicago Press, 1987

[94] JONES, James A. ; HARROLD, Mary J. ; STASKO, John: Visualization of test information to assist fault localization. In: *Proceedings of the 24th International Conference on Software Engineering*. New York, NY, USA : ACM, 2002 (ICSE '02), S. 467–477

[95] JUCKNATH-JOHN, Susanne ; GRAF, Dennis ; TAENTZER, Gabriele: Evolutionary layout: preserving the mental map during the development of class models. In: *SoftVis '06: Proceedings of the 2006 ACM symposium on Software visualization*. New York, NY, USA : ACM, 2006. – ISBN 1–59593–464–2, S. 165–166

[96] JUNGHANS, Martin: *Visualization of Hyperedges in Fixed Graph Layouts*, Brandenburg University of Technology, Cottbus, Germany, Diplomarbeit, 2008

[97] KAGDI, Huzefa H. ; HAMMAD, Maen ; MALETIC, Jonathan I.: Who can help me with this source code change? In: *Proceedings of the International Conference on Software Maintenance (ICSM '08)*. Washington, DC, USA : IEEE Computer Society, 2008, S. 157–166

[98] KERSTAN, Sven: *Visualisierung zeitbasierter Metriken in Software-Städten*. Bachelor Thesis, Brandenburg University of Technology, Cottbus, Germany, 2011

[99] KIENLE, Holger M. ; MULLER, Hausi A.: Requirements of Software Visualization Tools: A Literature Survey. In: *2007 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, IEEE, 2007, S. 2–9

[100] KNIGHT, Claire ; MUNRO, Malcolm: Comprehension with[in] Virtual Environment Visualisations. In: *Proceedings of the IEEE 7th International Workshop on Program Comprehension*. Washington, DC, USA : IEEE Computer Society, 1999, S. 4–11

[101] KNIGHT, Claire ; MUNRO, Malcolm: Virtual but Visible Software. In: *IV '00: Proceedings of the International Conference on Information Visualisation*. Washington, DC, USA : IEEE Computer Society, 2000. – ISBN 0–7695–0743–3, S. 198

[102] KOALICK, Sven: *Entwicklung eines Datenmodells und Faktenextraktors zur statischen Analyse von Software-Systemen unter Einbeziehung der Entwicklungshistorie*, Brandenburg University of Technology, Cottbus, Germany, Diplomarbeit, 2007

[103] KOCK, Christopher: *Integration von Versionskontrollinformationen in das Software Cockpit*. Bachelor Thesis, Brandenburg University of Technology, Cottbus, Germany, 2007

[104] KOPSCH, Ralf: *3D-Visualisierung von Projektverwaltungsinformationen und RT-Modellen*, Brandenburg University of Technology, Cottbus, Germany, Diplomarbeit, 2009

[105] KOSCHKE, Rainer: Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey. In: *Journal of Software Maintenance* 15 (2003), März, Nr. 2, 87–109. http://dx.doi.org/10.1002/smr.270. – DOI 10.1002/smr.270. – ISSN 1040–550X

[106] KOSSACK, Sven: *Integration des Visualisierungssystems Crococosmo in die Eclipse Entwicklungsumgebung*. Bachelor Thesis, Brandenburg University of Technology, Cottbus, Germany, 2010

[107] KUBOTA, Hidekazu ; NISHIDA, Toyoaki ; SUMI, Yasuyuki: Visualization of contents archive by contour map representation. In: *Proceedings of the 20th annual conference on New frontiers in artificial intelligence*. Berlin, Heidelberg : Springer-Verlag, 2007 (JSAI'06). – ISBN 978–3–540–69901–9, S. 19–32

[108] KUEPPERS, S. ; SCHIECK, A. Fatah g. ; MOTTRAM, C. ; PENN, A.: City metaphor based visualisation of collaborative work spaces in TOWER. In: *Space Syntax 5th International Symposium*, Techne Press, Amsterdam, 2005

[109] KUHN, Adrian ; ERNI, David ; LORETAN, Peter ; NIERSTRASZ, Oscar: Software Cartography: thematic software visualization with consistent layout. In: *Journal of Software Maintenance* 22 (2010), April, S. 191–210

[110] KUHN, Adrian ; LORETAN, Peter ; NIERSTRASZ, Oscar: Consistent Layout for Thematic Software Maps. In: *Proceedings of 15th Working Conference on Reverse Engineering (WCRE 2008)*, 2008, S. 209–218

[111] KUHN, Werner ; FRANK, Andrew U.: A Formalization Of Metaphors And Image-Schemas In User Interfaces. In: MARK, D.M. (Hrsg.) ; FRANK, A.U. (Hrsg.): *Cognitive and Linguistic Aspects of Geographic Space*, Kluwer Academic Publishers, Dordrecht, The Netherlands, 1991

[112] LAKOFF, George ; JOHNSON, Mark: *Metaphors we live by*. University of Chicago Press, 1980

[113] LANGE, Christian F. J. ; CHAUDRON, Michel R. V.: Interactive Views to Improve the Comprehension of UML Models - An Experimental Validation. In: *Proceedings of the 15th IEEE International Conference on Program Comprehension*. Washington, DC, USA : IEEE Computer Society, 2007. – ISBN 0–7695–2860–0, S. 221–230

[114] LANGE, Christian F. J. ; WIJNS, Martijn A. M. ; CHAUDRON, Michel R. V.: MetricViewEvolution: UML-based Views for Monitoring Model Evolution and Quality. In: *Proceedings of the 11th European Conference on Software Maintenance and Reengineering*. Washington, DC, USA : IEEE Computer Society, 2007. – ISBN 0–7695–2802–3, S. 327–328

[115] LANGELIER, Guillaume ; SAHRAOUI, Houari ; POULIN, Pierre: Visualization-based analysis of quality for large-scale software systems. In: *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. New York, NY, USA : ACM, 2005. – ISBN 1–59593–993–4, S. 214–223

[116] LANGELIER, Guillaume ; SAHRAOUI, Houari ; POULIN, Pierre: Exploring the evolution of software quality with animated visualization. In: *Proceedings of the 2008 IEEE Symposium on Visual Languages and Human-Centric Computing*. Washington, DC, USA : IEEE Computer Society, 2008 (VLHCC '08). – ISBN 978–1–4244–2528–0, S. 13–20

[117] LANKES, Josef ; MATTHES, Florian ; WITTENBURG, André: Softwarekartographie: Systematische Darstellung von Anwendungslandschaften. In: FERSTL, Otto K. (Hrsg.) ; SINZ, Elmar J. (Hrsg.) ; ECKERT, Sven (Hrsg.) ; ISSELHORST, Tilman (Hrsg.): *Wirtschaftsinformatik 2005*. Physica-Verlag HD, 2005. – ISBN 978–3–7908–1624–2, S. 1443–1462

[118] LANZA, Michele: The evolution matrix: recovering software evolution using software visualization techniques. In: *IWPSE '01: Proceedings of the 4th International Workshop on Principles of Software Evolution*. New York, NY, USA : ACM, 2001. – ISBN 1–58113–508–4, S. 37–42

[119] LANZA, Michele ; GALL, Harald ; DUGERDIL, Philip: EvoSpaces: Multi-dimensional Navigation Spaces for Software Evolution. In: *Proceedings of 13th IEEE European Conference on Software Maintenance and Reengineering*. Kaiserslautern, Germany : IEEE Computer Society, 2009

[120] LATOZA, Thomas D. ; VENOLIA, Gina ; DELINE, Robert: Maintaining mental models: a study of developer work habits. In: *ICSE '06: Proceedings of the 28th international conference on Software engineering*. New York, NY, USA : ACM, 2006. – ISBN 1–59593–375–1, S. 492–501

[121] LEGDE, Katharina: *Visualizing Test Data in Software Cities*. Bachelor Thesis, Brandenburg University of Technology, Cottbus, Germany, 2011

[122] LEWERENTZ, Claus ; NOACK, Andreas: CrocoCosmos - 3D Visualization of Large Object Oriented Programs. In: MUTZEL, Petra (Hrsg.) ; JÜNGER, Michael (Hrsg.): *Graph Drawing Software*. Springer Verlag, 2003. – ISBN 3–540–00881–0, S. 279–297

[123] LEWERENTZ, Claus ; SIMON, Frank: Metrics-Based 3D Visualization of Large Object-Oriented Programs. In: *VISSOFT '02: Proceedings of the 1st International Workshop on Visualizing Software for Understanding and Analysis*. Washington, DC, USA : IEEE Computer Society, 2002. – ISBN 0–7695–1662–9, S. 70

[124] LEWERENTZ, Claus ; SIMON, Frank ; STEINBRÜCKNER, Frank: CrocoCosmos. In: MUTZEL, Petra (Hrsg.) ; JÜNGER, Michael (Hrsg.) ; LEIPERT, Sebastian (Hrsg.): *Graph Drawing* Bd. 2265. Springer Berlin / Heidelberg, 2002. – ISBN 978–3–540–43309–5, S. 72–76

[125] LEWERENTZ, Claus ; STEINBRÜCKNER, Frank: SoftUrbs: Visualizing Software Systems as Urban Structures / Computer Science Reports 02/2009, BTU Cottbus. 2009. – Forschungsbericht

[126] LÜ, Hao ; FOGARTY, James: Cascaded treemaps: examining the visibility and stability of structure in treemaps. In: *Proceedings of graphics interface 2008*. Toronto, Ont., Canada, Canada : Canadian Information Processing Society, 2008 (GI '08). – ISBN 978–1–56881–423–0, S. 259–266

[127] LYONS, Kelly A. ; MEIJER, Henk ; RAPPAPORT, David: Algorithms for Cluster Busting in Anchored Graph Drawing. In: *Journal of Graph Algorithms and Applications*, 1998, S. 7–17

[128] MAAG, Volker ; BERGER, Martin ; WINTERFELD, Anton ; KÜFER, Karl-Heinz: A novel non-linear approach to minimal area rectangular packing. In: *Annals of Operations Research*

[129] MACKINLAY, Jock: Automating the design of graphical presentations of relational information. In: *ACM Transactions on Graphics* 5 (1986), April, S. 110–141. – ISSN 0730–0301

[130] MALETIC, Jonathan I. ; MARCUS, Andrian ; COLLARD, Michael L.: A Task Oriented View of Software Visualization. In: *VISSOFT '02: Proceedings of the 1st International Workshop on Visualizing Software for Understanding and Analysis*. Washington, DC, USA : IEEE Computer Society, 2002. – ISBN 0–7695–1662–9, S. 32

[131] MALLOY, Brian A. ; POWER, James F.: Using a Molecular Metaphor to Facilitate Comprehension of 3D Object Diagrams. In: *Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*. Washington, DC, USA : IEEE Computer Society, 2005. – ISBN 0–7695–2443–5, 233–240

[132] MANNL, Uwe: *Evaluation of layout stability of software cities*, Brandenburg University of Technology, Cottbus, Germany, Diplomarbeit, 2010

[133] MARCUS, Andrian ; FENG, Louis ; MALETIC, Jonathan I.: 3D representations for software visualization. In: *SoftVis '03: Proceedings of the 2003 ACM symposium on Software visualization*. New York, NY, USA : ACM, 2003. – ISBN 1–58113–642–0, S. 27 ff

[134] MARCUS, Andrian ; FENG, Louis ; MALETIC, Jonathan I.: Comprehension of Software Analysis Data Using 3D Visualization. In: *IWPC '03: Proceedings of the 11th IEEE International Workshop on Program Comprehension*. Washington, DC, USA : IEEE Computer Society, 2003. – ISBN 0–7695–1883–4, S. 105

[135] MCCABE, Thomas J.: A Complexity Measure. In: *IEEE Transactions on Software Engineering* 2 (1976), S. 308–320. – ISSN 0098–5589

[136] MCDONALD, David W.: Evaluating expertise recommendations. In: *GROUP '01: Proceedings of the 2001 International ACM SIGGROUP Conference on Supporting Group Work*. New York, NY, USA : ACM, 2001. – ISBN 1–58113–294–8, S. 214–223

[137] MCDONALD, David W. ; ACKERMAN, Mark S.: Expertise recommender: a flexible recommendation system and architecture. In: *Proceedings of the 2000 ACM conference on Computer supported cooperative work (CSCW '00)*. New York, NY, USA : ACM, 2000. – ISBN 1–58113–222–0, S. 231–240

[138] MINTO, Shawn ; MURPHY, Gail C.: Recommending Emergent Teams. In: *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*. Washington, DC, USA : IEEE Computer Society, 2007, S. 5

[139] MISUE, Kazuo ; EADES, Peter ; LAI, Wei ; SUGIYAMA, Kozo: Layout Adjustment and the Mental Map. In: *Journal of Visual Languages and Computing* 6 (1995), Nr. 2, S. 183–210

[140] MOCKUS, Audris ; HERBSLEB, James D.: Expertise browser: a quantitative approach to identifying expertise. In: *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*. New York, NY, USA : ACM, 2002, S. 503–512

[141] MYERS, Brad A.: Visual programming, programming by example, and program visualization: a taxonomy. In: *SIGCHI Bull.* 17 (1986), Nr. 4, S. 59–66. – ISSN 0736–6906

[142] MYERS, Brad A.: Taxonomies of visual programming and program visualization. In: *Journal of Visual Languages and Computing* 1 (1990), March, S. 97–123. – ISSN 1045–926X

[143] NOACK, Andreas ; LEWERENTZ, Claus: A space of layout styles for hierarchical graph models of software systems. In: *Proceedings of the 2005 ACM symposium on Software visualization*. New York, NY, USA : ACM, 2005 (SoftVis '05). – ISBN 1–59593–073–6, 155–164

[144] NORTH, Stephen: Incremental layout in DynaDAG. In: BRANDENBURG, Franz (Hrsg.): *Graph Drawing* Bd. 1027. Springer Berlin / Heidelberg, 1996, S. 409–418

[145] PANAS, Thomas ; BERRIGAN, Rebecca ; GRUNDY, John: A 3D Metaphor for Software Production Visualization. In: *IV '03: Proceedings of the Seventh International Conference on Information Visualization*. Washington, DC, USA : IEEE Computer Society, 2003. – ISBN 0–7695–1988–1, S. 314

[146] PANAS, Thomas ; EPPERLY, Thomas ; QUINLAN, Daniel ; SÆBJØRNSEN, Andreas ; VUDUC, Richard: Communicating Software Architecture using a Unified Single-View Visualization. In: *Proceedings of IEEE International Conference on Engineering of Complex Computer Systems*. Los Alamitos, CA, USA : IEEE Computer Society, 2007, S. 217–228

[147] PETRE, Marian ; QUINCEY, Ed de: A gentle overview of software visualisation, Computer Society of India Communications, 2006

[148] PLOIX, D.: Analogical Representations of Programs. In: *VISSOFT '02: Proceedings of the 1st International Workshop on Visualizing Software for Understanding and Analysis*. Washington, DC, USA : IEEE Computer Society, 2002. – ISBN 0–7695–1662–9, S. 61

[149] PRICE, Blaine ; BAECKER, Ronald ; SMALL, Ian: An introduction to software visualization. In: STASKO, John (Hrsg.) ; DOMINGUE, John (Hrsg.) ; BROWN, Marc H. (Hrsg.) ; PRICE, Blaine A. (Hrsg.): *Software Visualization, Programming as a Multimedia experience*, The MIT Press, 1997. – ISBN 0–262–19395–7

[150] PRICE, Blaine A. ; SMALL, Ian S. ; BAECKER, Ronald M.: A taxonomy of software visualization. In: *Proceedings of Twenty-Fifth Hawaii International Conference on System Sciences*, 1992, S. 597–606

[151] PURCHASE, Helen ; HOGGAN, Eve ; GÖRG, Carsten: How Important Is the Mental Map? - An Empirical Investigation of a Dynamic Graph Layout Algorithm. In: KAUFMANN, Michael (Hrsg.) ; WAGNER, Dorothea (Hrsg.): *Graph Drawing* Bd. 4372. Springer Berlin / Heidelberg, 2007, S. 184–195

[152] RÖDER, Stephanie: *Integration des Visualisierungssystems Crococosmo in die Eclipse Entwicklungsumgebung*. Bachelor Thesis, Brandenburg University of Technology, Cottbus, Germany, 2011

[153] REISS, S. P.: The Paradox of Software Visualization. In: *Proceedings of the 3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis*. Washington, DC, USA : IEEE Computer Society, 2005 (VISSOFT '05). – ISBN 0–7803–9540–9, S. 19 ff

[154] REISS, Steven P.: Bee/Hive: A Software Visualization Back End. In: *Proceedings of ICSE 2011 Workshop on Software Visualization*, 2001, S. 44–48

[155] REUSSNER, Ralf H. ; HASSELBRING, Wilhelm: *Handbuch der Software-Architektur*. dPunkt.verlag Heidelberg, 2006. – ISBN 3–89864–372–7

[156] ROMAN, Gruia-Catalin ; COX, Kenneth C.: Program visualization: the art of mapping programs to pictures. In: *ICSE '92: Proceedings of the 14th international conference on Software engineering*. New York, NY, USA : ACM, 1992. – ISBN 0–89791–504–6, S. 412–420

[157] SAFFREY, Peter ; PURCHASE, Helen: The "mental map" versus "static aesthetic" compromise in dynamic graphs: a user study. In: *Proceedings of the ninth conference on Australasian user interface - Volume 76*. Darlinghurst, Australia, Australia : Australian Computer Society, Inc., 2008 (AUIC '08). – ISBN 978–1–920682–57–6, S. 85–93

[158] SANTOS, C. Russo D. ; GROS, P. ; ABEL, P. ; LOISEL, D. ; TRICHAUD, N. ; PARIS, J.P.: Mapping Information onto 3D Virtual Worlds. In: *Proceedings of International Conference on Information Visualisation* 0 (2000), S. 379. – ISSN 1093–9547

[159] SCHRECK, Tobias ; KEIM, Da ; MANSMANN, Florian: Regular TreeMap Layouts for Visual Analysis of Hierarchical Data. In: *Proceedings of the Spring Conference on Computer Graphics (SCCG06)*, Comenius University, Bratislava, 2006

[160] SCHULZ, Hans-Jorg ; HADLAK, Steffen ; SCHUMANN, Heidrun: The Design Space of Implicit Hierarchy Visualization: A Survey. In: *IEEE Transactions on Visualization and Computer Graphics* 17 (2011), S. 393–411. – ISSN 1077–2626

[161] SCHUMANN, H. ; MÜLLER, W.: *Visualisierung: Grundlagen und allgemeine Methoden*. Springer-Verlag New York Incorporated, 2000. – ISBN 9783540649441

[162] SENSALIRE, Mariam ; OGAO, Patrick: Tool users requirements classification: how software visualization tools measure up. In: *Proceedings of the 5th international conference on Computer graphics, virtual reality, visualisation and interaction in Africa*. New York, NY, USA : ACM, 2007 (AFRIGRAPH '07). – ISBN 978–1–59593–906–7, S. 119–124

[163] SENSALIRE, Mariam ; OGAO, Patrick ; TELEA, Alexandru: Classifying desirable features of software visualization tools for corrective maintenance. In: *Proceedings of the 4th ACM symposium on Software visualization*. New York, NY, USA : ACM, 2008 (SoftVis '08). – ISBN 978–1–60558–112–5, S. 87–90

[164] SHNEIDERMAN, Ben: Tree visualization with tree-maps: 2-d space-filling approach. In: *ACM Transactions on Graphics* 11 (1992), January, S. 92–99. – ISSN 0730–0301

[165] SHNEIDERMAN, Ben: The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations. In: *Visual Languages, IEEE Symposium on* 0 (1996), S. 336. – ISSN 1049–2615

[166] SIEDERSLEBEN, Johannes: *Moderne Softwarearchitektur - umsichtig planen, robust bauen mit Quasar*. dpunkt.verlag, 2004. – ISBN 978–3–89864–292–7

[167] SIMON, Frank ; STEINBRÜCKNER, Frank ; LEWERENTZ, Claus: Metrics Based Refactoring. In: *CSMR '01: Proceedings of the Fifth European Conference on Software Maintenance and Reengineering*. Washington, DC, USA : IEEE Computer Society, 2001. – ISBN 0–7695–1028–0, S. 30

[168] SKUPIN, Andre ; BUTTENFIELD, Barbara P.: Spatial Metaphors for Visualizing Information Spaces. In: *ACSM/ASPRS Annual Convention and Exhibition Seattle,WA April 7-10,Technical Papers*, 1997, S. 116–125

[169] SPARACINO, Flavia ; WREN, Christopher R. ; AZARBAYEJANI, Ali ; PENTLAND, Alex: Browsing 3-D spaces with 3-D vision: body-driven navigation through the Internet city. In: *1st International Symposium on 3D Data Processing Visualization and Transmission*, 2002, S. 224–233

[170] SPENCE, Robert: *Information visualization*. Harlow, England : Addison-Wesley, 2001 (ACM Press books). – ISBN 0–201–59626–1

[171] STASKO, John (Hrsg.) ; DOMINGUE, John (Hrsg.) ; BROWN, Marc H. (Hrsg.) ; PRICE, Blaine A. (Hrsg.): *Software Visualization, Programming as a Multimedia Experience*. MIT Press, Cambridge, MA, 1998

[172] STASKO, John T. ; PATTERSON, Charles: Understanding and Characterizing Software Visualization Systems. In: *Proceedings of the 1992 IEEE Workshop on Visual Languages, September 15-18, 1992, Seattle, Washington, USA*, IEEE Computer Society. – ISBN 0–8186–3090–6

[173] STEINBRÜCKNER, Frank ; LEWERENTZ, Claus: Representing development history in software cities. In: *Proceedings of the 5th international symposium on Software visualization*. New York, NY, USA : ACM, 2010 (SOFTVIS '10). – ISBN 978–1–4503–0028–5, S. 193–202

[174] TAUER, Michael: *Design, realisation and evaluation of an approach to visualize changing decompositions*, Brandenburg University of Technology, Cottbus, Germany, Diplomarbeit, 2010

[175] TERMEER, M. ; LANGE, C. F. J. ; TELEA, A. ; CHAUDRON, M. R. V.: Visual Exploration of Combined Architectural and Metric Information. In: *Proceedings of the 3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis*. Washington, DC, USA : IEEE Computer Society, 2005 (VISSOFT '05). – ISBN 0–7803–9540–9, 11 ff

[176] TEYSEYRE, Alfredo R. ; CAMPO, Marcelo R.: An Overview of 3D Software Visualization. In: *IEEE Transactions on Visualization and Computer Graphics* 15 (2009), January, S. 87–105. – ISSN 1077–2626

[177] TÜLLING, Steffen: *Verbesserung eines Layoutansatzes für Software-Landschaften*, Brandenburg University of Technology, Cottbus, Germany, Diplomarbeit, 2012

[178] TU, Qiang ; GODFREY, Michael W.: An Integrated Approach for Studying Architectural Evolution. In: *Proceedings of the 10th International Workshop on Program Comprehension*. Washington, DC, USA : IEEE Computer Society, 2002 (IWPC '02). – ISBN 0–7695–1495–2, S. 127 ff

[179] TU, Ying ; SHEN, Han-Wei: Visualizing Changes of Hierarchical Data using Treemaps. In: *IEEE Transactions on Visualization and Computer Graphics* 13 (2007), November, S. 1286–1293. – ISSN 1077–2626

[180] TUFTE, Edward R.: *The visual display of quantitative information*. 2. Aufl. Graphics Press, 2001. – ISBN 0961392142

[181] TURO, David ; JOHNSON, Brian: Improving the visualization of hierarchies with treemaps: design issues and experimentation. In: *Proceedings of the 3rd conference on Visualization '92*. Los Alamitos, CA, USA : IEEE Computer Society Press, 1992 (VIS '92). – ISBN 0–8186–2896–0, S. 124–131

[182] UHLIG, Marcus: *Entwicklung und Evaluation eines Edge-Routingverfahrens für Softwarelandschaften*. Bachelor Thesis, Brandenburg University of Technology, Cottbus, Germany, 2012

[183] UHLIG, Markus: *Visualisierung von Konzepten in Software-Städten*, Brandenburg University of Technology, Cottbus, Germany, Diplomarbeit, 2012

[184] VAN WIJK, Jarke J. ; WETERING, Huub van d.: Cushion Treemaps: Visualization of Hierarchical Information. In: *Proceedings of the 1999 IEEE Symposium on Information Visualization*. Washington, DC, USA : IEEE Computer Society, 1999. – ISBN 0–7695–0431–0, S. 73 ff

[185] ČUBRANIĆ, Davor ; MURPHY, Gail C.: Hipikat: recommending pertinent software development artifacts. In: *Proceedings of the 25th International Conference on Software Engineering*. Washington, DC, USA : IEEE Computer Society, 2003 (ICSE '03). – ISBN 0–7695–1877–X, S. 408–418

[186] VOINEA, Stefan-Lucian: *Software Evolution Visualization*, Technische Universiteit Eindhoven, Netherlands, Diss., 2007

[187] WATTENBERG, Martin: Visualizing the stock market. In: *CHI '99 extended abstracts on Human factors in computing systems*, ACM, 1999 (CHI EA '99). – ISBN 1–58113–158–5, S. 188–189

[188] WATTENBERG, Martin: A Note on Space-Filling Visualizations and Space-Filling Curves. In: *Proceedings of the 2005 IEEE Symposium on Information Visualization*. Washington, DC, USA : IEEE Computer Society, 2005. – ISBN 0–7803–9464–x

[189] WEISSGERBER, Peter ; POHL, Mathias ; BURCH, Michael: Visual Data Mining in Software Archives to Detect How Developers Work Together. In: *Proceedings of the Fourth International Workshop on Mining Software Repositories*. Washington, DC, USA : IEEE Computer Society, 2007 (MSR '07). – ISBN 0–7695–2950–X, S. 9 ff

[190] WENSIECKI, Sven: *Animation der Entwicklung von Strukturen und Kennzahlen von Software-Systemen*, Brandenburg University of Technology, Cottbus, Germany, Diplomarbeit, 2009

[191] WERNER, Erik: *Software-Visualisierung mit Hilfe eines Kartendienstes*. Bachelor Thesis, Brandenburg University of Technology, Cottbus, Germany, 2011

[192] WETTEL, Richard: *Software Systems as Cities*, University of Lugano, Switzerland, Diss., 2010

[193] WETTEL, Richard ; LANZA, Michele: Program Comprehension through Software Habitability. In: *ICPC '07: Proceedings of the 15th IEEE International Conference on Program Comprehension*. Washington, DC, USA : IEEE Computer Society, 2007. – ISBN 0–7695–2860–0, S. 231–240

[194] WETTEL, Richard ; LANZA, Michele: Visualizing Software Systems as Cities. In: *Proceedings of VISSOFT 2007 (4th International Workshop on Visualizing Software For Understanding and Analysis*, IEEE Computer Society, 2007, S. 92–99

[195] WETTEL, Richard ; LANZA, Michele: Visual Exploration of Large-Scale System Evolution. In: *WCRE '08: Proceedings of the 2008 15th Working Conference on Reverse Engineering*. Washington, DC, USA : IEEE Computer Society, 2008. – ISBN 978–0–7695–3429–9, S. 219–228

[196] WETTEL, Richard ; LANZA, Michele: Visually localizing design problems with disharmony maps. In: *SoftVis '08: Proceedings of the 4th ACM symposium on Software visualization*. New York, NY, USA : ACM, 2008. – ISBN 978–1–60558–112–5, S. 155–164

[197] WETTEL, Richard ; LANZA, Michele ; ROBBES, Romain: Software systems as cities: a controlled experiment. In: *Proceedings of the 33rd International Conference on Software Engineering*. New York, NY, USA : ACM, 2011 (ICSE '11). – ISBN 978–1–4503–0445–0, S. 551–560

[198] WINKEL, Christian: *Entwurf, Implementierung und Evaluation eines Sunburst basierten Visualisierungsansatzes für veränderliche Dekompositionsstrukturen*. Bachelor Thesis, Brandenburg University of Technology, Cottbus, Germany, 2011

[199] WISE, J. A. ; THOMAS, J. J. ; PENNOCK, K. ; LANTRIP, D. ; POTTIER, M. ; SCHUR, A. ; CROW, V.: Visualizing the non-visual: spatial analysis and interaction with information from text documents. In: *Proceedings of the 1995 IEEE Symposium on Information Visualization*. Washington, DC, USA : IEEE Computer Society, 1995. – ISBN 0–8186–7201–3, S. 51 ff

[200] WOODS, Eoin ; ROZANSKI, Nick: Unifying software architecture with its implementation. In: *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume*. New York, NY, USA : ACM, 2010 (ECSA '10). – ISBN 978–1–4503–0179–4, S. 55–58

[201] WU, Jingwei ; SPITZER, Claus W. ; HASSAN, Ahmed E. ; HOLT, Richard C.: Evolution Spectrographs: Visualizing Punctuated Change in Software Evolution. In: *Proceedings of the Principles of Software Evolution, 7th International Workshop*. Washington, DC, USA : IEEE Computer Society, 2004. – ISBN 0–7695–2211–4, S. 57–66

[202] XIE, Xinrong ; POSHYVANYK, Denys ; MARCUS, Adrian: Support for Static Concept Location with sv3D. In: *VISSOFT '05: Proceedings of the 3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis*. Washington, DC, USA : IEEE Computer Society, 2005. – ISBN 0–7803–9540–9, S. 28

[203] XIE, Xinrong ; POSHYVANYK, Denys ; MARCUS, Andrian: 3D visualization for concept location in source code. In: *ICSE '06: Proceedings of the 28th international conference on Software engineering*. New York, NY, USA : ACM, 2006. – ISBN 1–59593–375–1, S. 839–842

[204] YOUNG, P. ; MUNRO, M.: Visualizing Software in Virtual Reality. In: *International Conference on Program Comprehension*. Los Alamitos, CA, USA : IEEE Computer Society, 1998, S. 19

[205] YU, Liguo ; RAMASWAMY, Srini: Mining CVS Repositories to Understand Open-Source Project Developer Roles. In: *Proceedings of the Fourth International Workshop on Mining Software Repositories*. Washington, DC, USA : IEEE Computer Society, 2007 (MSR '07). – ISBN 0–7695–2950–X, 8 ff

[206] ZIEBE, Marco: *Verknüpfung von Prozess- und Produktmetriken*. Bachelor Thesis, Brandenburg University of Technology, Cottbus, Germany, 2009

[207] ZIERENBERG, Marcel: *Visualizing runtime-data with Software-Cities*, Brandenburg University of Technology, Cottbus, Germany, Diplomarbeit, 2010

Appendix A

Data Sets

For the following software systems, primary models were extracted and analyzed by revision from their version control systems.

System:	Apache Ant
Location:	http://archive.apache.org/dist/ant/binaries/
Revisions:	1.1, 1.2, 1.3, 1.4.0, 1.4.1, 1.5.0, 1.5.1, 1.5.2, 1.5.3, 1.5.4, 1.6.0, 1.6.1, 1.6.2, 1.6.3, 1.6.4, 1.6.5, 1.7.0, 1.7.1, 1.8.0-0RC1, 1.8.0-1F, 1.8.1, 1.8.2
Restrictions:	org.apache.tools.ant

System:	Apache CXF
Location:	http://archive.apache.org/dist/cxf/
Revisions:	2.0.06, 2.0.07, 2.0.08, 2.0.09, 2.0.10, 2.0.11, 2.0.12, 2.0.13, 2.1.00, 2.1.01, 2.1.02, 2.1.03, 2.1.04, 2.1.05, 2.1.06, 2.1.07, 2.1.08, 2.1.09, 2.1.10, 2.2.00, 2.2.01, 2.2.02, 2.2.03, 2.2.04, 2.2.05, 2.2.06, 2.2.07, 2.2.08, 2.2.09, 2.2.10, 2.2.11, 2.2.12, 2.3.00, 2.3.01, 2.3.02, 2.3.03, 2.3.04, 2.3.05, 2.3.06, 2.4.0, 2.4.1, 2.4.2
Restrictions:	org.apache.cxf

System:	Apache Derby
Location:	http://db.apache.org/derby/derby_downloads.html
Revisions:	10.1.1.0, 10.1.2.1, 10.1.3.1, 10.2.1.6, 10.2.2.0, 10.3.3.0, 10.4.1.3, 10.4.2.0, 10.5.1.1, 10.5.3.0, 10.6.1.0, 10.6.2.1, 10.7.1.1
Restrictions:	-

System:	ArgoUML
Location:	http://argouml-downloads.tigris.org/
Revisions:	0.10.1, 0.12, 0.14, 0.16, 0.18.1, 0.20, 0.22, 0.24, 0.26, 0.26.2, 0.28, 0.28.1, 0.30, 0.30.1, 0.30.2, 0.32, 0.32.1, 0.32.2
Restrictions:	-

System	# Versions	# $C_{initial}$	# C_{final}	Source
Apache Ant	22	94	1044	Releases
Apache CXF	42	1490	2760	Releases
Apache Derby	13	1198	1494	Releases
ArgoUML	18	908	1921	Releases
AspectJTools	13	3500	3527	Releases
Checkstyle	26	23	692	Versions
Clojure	15	314	753	Releases
Compass	20	1174	1539	Releases
CrocoCosmo	51	389	509	Versions
Datanucleous Core	48	612	849	Releases
DP TTPB	5	451	760	Releases
DP AN	5	241	1578	Releases
FindBugs	22	67	1024	Versions
Hibernate.Core	43	1374	2696	Releases
HSQLDB	14	57	543	Releases
JFreeChart	51	90	611	Releases
JME3	19	736	898	Versions
JMol	22	542	748	Releases
MMI3G	8	4224	5258	Releases
Mule.Core	43	564	1131	Releases
Neo4J	26	364	561	Releases
OpenSAML	8	101	1410	Releases
PMD	15	118	687	Versions
ProcessDash	37	55	925	Versions
Spring Framework	49	708	2150	Releases

Table A.1: Example Software Systems

System:	AspectJTools
Location:	Maven Central Repository (search.maven.org) /central/org/aspectj/aspectjtools)
Revisions:	1.5.4, 1.6.00, 1.6.01, 1.6.02, 1.6.03, 1.6.04, 1.6.05, 1.6.06, 1.6.07, 1.6.08, 1.6.09, 1.6.10, 1.6.11
Restrictions:	org

System:	Checkstyle
Location:	https://checkstyle.svn.sourceforge.net/svnroot/checkstyle/trunk
Revisions:	10, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 1100, 1200, 1300, 1400, 1500, 1600, 1700, 1800, 1900, 2000, 2100, 2200, 2300, 2400, 2500
Restrictions:	com.puppycrawl.tools.checkstyle

System:	Clojure
Location:	Maven Central Repository (search.maven.org) /central/org/clojure/clojure
Revisions:	1.0.0, 1.1.0, 1.2.0, 1.2.1, 1.3.0, 1.3.0-RC0, 1.3.0-alpha5, 1.3.0-alpha6, 1.3.0-alpha7, 1.3.0-alpha8, 1.3.0-beta1, 1.3.0-beta2, 1.3.0-beta3, 1.4.0-alpha1, 1.4.0-alpha2
Restrictions:	org.compass

System:	Compass
Location:	Maven Central Repository (search.maven.org) /central/org/compass-project/compass
Revisions:	1.2.2, 2.0.0-RC2, 2.0.0, 2.0.1, 2.0.2, 2.1.0-M1, 2.1.0-M2, 2.1.0-M3, 2.1.0-M4, 2.1.0-RC, 2.1.0, 2.1.1, 2.1.2, 2.1.3, 2.1.4, 2.2.0-M1, 2.2.0-M2, 2.2.0-RC1, 2.2.0-RC2, 2.2.0
Restrictions:	org.compass

System:	CrocoCosmo
Location:	https://svn.informatik.tu-cottbus.de/~fsteinbr/dev/CrocoCosmo
Revisions:	11, 20, 40, 60, 80, 100, 120, 140, 160, 180, 200, 220, 240, 260, 280, 300, 320, 340, 360, 380, 400, 420, 440, 460, 480, 500, 520, 540, 560, 580, 600, 620, 640, 660, 680, 700, 720, 740, 760, 780, 800, 820, 840, 860, 880, 900, 920, 940, 960, 980, 1000

System:	Datanucleus Core
Location:	Maven Central Repository (search.maven.org) /central/org/datanucleus/datanucleus-core
Revisions:	1.0.2, 1.0.3, 1.0.4, 1.0.5, 1.1.0.m2, 1.1.0.m3, 1.1.0.m4, 1.1.0, 1.1.1, 1.1.2, 1.1.3, 1.1.4, 1.1.5, 1.1.6, 2.0.0.m1, 2.0.0-m2, 2.0.0-m3, 2.0.0-m4, 2.0.0-release, 2.0.1, 2.0.2, 2.0.3, 2.0.4, 2.0.5, 2.1.0-m1, 2.1.0-m2, 2.1.0-m3, 2.1.0-release, 2.1.1, 2.1.2, 2.1.3, 2.1.4, 2.2.0-m1, 2.2.0-m2, 2.2.0-m3, 2.2.0-release, 2.2.1, 2.2.2, 2.2.3, 2.2.4, 3.0.0-m1, 3.0.0-m2, 3.0.0-m3, 3.0.0-m4, 3.0.0-m5, 3.0.0-m6, 3.0.0-release, 3.0.1
Restrictions:	-

System:	FindBugs
Location:	http://findbugs.googlecode.com/svn/trunk/findbugs
Revisions:	500, 1000, 1500, 2000, 2500, 3000, 3500, 4000, 4500, 5000, 5500, 6000, 6500, 7000, 7500, 8000, 8500, 9000, 9500, 10000, 10500, 11000
Restrictions:	edu.umd.cs.findbugs

System:	Hibernate Core
Location:	Maven Central Repository (search.maven.org) /central/org/hibernate/hibernate-core/
Revisions:	3.3.0.CR1, 3.3.0.CR2, 3.3.0.GA, 3.3.0.SP1, 3.3.1.GA, 3.3.2.GA, 3.5.0-Beta-2, 3.5.0-Beta-3, 3.5.0-Beta-4, 3.5.0-CR-1, 3.5.0-CR-2, 3.5.0-Final, 3.5.0.Beta-1, 3.5.1-Final, 3.5.2-Final, 3.5.3-Final, 3.5.4-Final, 3.5.5-Final, 3.5.6-Final, 3.6.0.Beta1, 3.6.0.Beta2, 3.6.0.Beta3, 3.6.0.Beta4, 3.6.0.CR1, 3.6.0.CR2, 3.6.0.Final, 3.6.1.Final, 3.6.2.Final, 3.6.3.Final, 3.6.4.Final, 3.6.5.Final, 3.6.6.Final, 3.6.7.Final, 4.0.0.Alpha1, 4.0.0.Alpha2, 4.0.0.Alpha3, 4.0.0.Beta1, 4.0.0.Beta2, 4.0.0.Beta3, 4.0.0.Beta4, 4.0.0.Beta5, 4.0.0.CR1, 4.0.0.CR2
Restrictions:	-

System:	HSQLDB
Location:	http://sourceforge.net/projects/hsqldb/files/hsqldb/
Revisions:	1.6.1, 1.7.0, 1.7.1, 1.7.2.11, 1.7.3.3, 1.8.0.10, 1.8.1.3, 2.0.0, 2.2.0, 2.2.1, 2.2.2, 2.2.3, 2.2.4, 2.2.5
Restrictions:	-

System:	JFreeChart
Location:	“http://sourceforge.net/projects/jfreechart/files/1. JFreeChart/”
Revisions:	0.5.60, 0.6.00, 0.7.00, 0.7.01, 0.7.02, 0.7.03, 0.7.04, 0.8.00, 0.8.01, 0.9.00, 0.9.01, 0.9.02, 0.9.03, 0.9.04, 0.9.05, 0.9.06, 0.9.07, 0.9.08, 0.9.09, 0.9.10, 0.9.11, 0.9.12, 0.9.13, 0.9.14, 0.9.15, 0.9.16, 0.9.17, 0.9.18, 0.9.19, 0.9.20, 0.9.21, 1.0.0-1pre1, 1.0.0-2pre2, 1.0.0-3rc1, 1.0.0-4rc2, 1.0.0-5rc3, 1.0.0-final, 1.0.01, 1.0.02, 1.0.03, 1.0.04, 1.0.05, 1.0.06, 1.0.07, 1.0.08, 1.0.08a, 1.0.09, 1.0.10, 1.0.11, 1.0.12, 1.0.13
Restrictions:	-

System:	JME 3
Location:	http://jmonkeyengine.com/nightly/
Revisions:	Nightly built archives: 2011-03-16, 2011-03-16, 2011-03-23, 2011-03-23, 2011-04-13, 2011-04-13, 2011-04-28, 2011-04-28, 2011-05-12, 2011-05-12, 2011-05-26, 2011-05-26, 2011-06-09, 2011-06-09, 2011-06-23, 2011-06-23, 2011-07-07, 2011-07-07, 2011-07-21, 2011-07-21, 2011-08-04, 2011-08-04, 2011-08-19, 2011-08-19, 2011-08-26, 2011-08-26, 2011-09-17, 2011-09-17, 2011-10-01, 2011-10-01, 2011-10-16, 2011-10-16, 2011-10-30, 2011-10-30, 2011-11-14, 2011-11-14, 2011-11-28, 2011-11-28
Restrictions:	com.jme3

System:	JMol
Location:	http://sourceforge.net/projects/jmol/files/Jmol/
Revisions:	11.8.1, 11.8.5, 11.8.9, 11.8.13, 11.8.17, 11.8.21, 11.8.25, 12.0.3, 12.0.7, 12.0.11, 12.0.15, 12.0.19, 12.0.23, 12.0.27, 12.0.31, 12.0.35, 12.0.39, 12.0.43, 12.0.47, 12.2.0, 12.2.4, 12.2.7
Restrictions:	org.jmol

System:	Mule Core
Location:	Maven Central Repository (search.maven.org) /central/org/mule/mule-core/
Revisions:	1.3-rc5, 1.3.1, 1.3.2, 1.3.3, 1.3, 1.4-RC1, 1.4.0, 1.4.1, 1.4.2, 1.4.3, 1.4.4, 2.0-M1, 2.0.0-M2, 2.0.0-RC1, 2.0.0-RC2, 2.0.0-RC3, 2.0.0, 2.0.1, 2.0.2, 2.1.0-M2, 2.1.0, 2.1.1, 2.1.2, 2.2.0, 2.2.1, 3.0.0-M1, 3.0.0-M2-20090803, 3.0.0-M2-20091006, 3.0.0-M2-20091026, 3.0.0-M2-20091124, 3.0.0-M2-20091130, 3.0.0-M2, 3.0.0-M3, 3.0.0-M4, 3.0.0-RC1, 3.0.0-RC2, 3.0.0, 3.0.1, 3.1.0-RC1, 3.1.0, 3.1.1, 3.1.2, 3.2.0-M1
Restrictions:	org.mule

System:	Neo4J
Location:	Maven Central Repository (search.maven.org) /central/org/neo4j/neo4j/
Revisions:	1.2.M01, 1.2.M02, 1.2.M03, 1.2.M04, 1.2.M05, 1.2.M06, 1.2, 1.3.M01, 1.3.M02, 1.3.M03, 1.3.M04, 1.3.M05, 1.3, 1.4.1, 1.4.2, 1.4.M01, 1.4.M02, 1.4.M03, 1.4.M04, 1.4.M05, 1.4.M06, 1.4, 1.5.M01, 1.5.M02, 1.5, 1.6.M01
Restrictions:	-

System:	OpenSAML
Location:	Maven Central Repository (search.maven.org) /central/org/opensaml/opensaml/
Revisions:	1.1, 2.1.0, 2.2.0, 2.2.1, 2.2.3, 2.4.1, 2.5.1-1, 2.5.1
Restrictions:	-

System:	PMD
Location:	https://pmd.svn.sourceforge.net/svnroot/pmd/trunk/pmd
Revisions:	20, 500, 1000, 1500, 2000, 2500, 3000, 3500, 4000, 4500, 5000, 5500, 6000, 6500, 7000
Restrictions:	-

System:	Process Dashboard
Location:	https://processdash.svn.sourceforge.net/svnroot/processdash /trunk/processdash
Revisions:	2000, 2050, 2100, 2150, 2200, 2250, 2300, 2350, 2400, 2450, 2500, 2550, 2600, 2650, 2700, 2750, 2800, 2850, 2900, 2950, 3000, 3050, 3100, 3150, 3200, 3250, 3300, 3350, 3400, 3450, 3500, 3550, 3600, 3650, 3700, 3750, 3800

System:	Spring Framework
Location:	Maven Central Repository (search.maven.org) /central/org/springframework/spring/
Revisions:	1.0-m4, 1.0-rc1, 1.0, 1.1-rc1, 1.1-rc2, 1.1, 1.1.1, 1.1.2, 1.1.3, 1.1.4, 1.1.5, 1.2-rc1, 1.2-rc2, 1.2, 1.2.1, 1.2.2, 1.2.3, 1.2.4, 1.2.5, 1.2.6, 1.2.7, 1.2.8, 1.2.9, 2.0-m1, 2.0-m2, 2.0-m3, 2.0-m4, 2.0-m5, 2.0-rc1, 2.0-rc2, 2.0-rc3, 2.0, 2.0.1, 2.0.2, 2.0.3, 2.0.4, 2.0.5, 2.0.6, 2.0.7, 2.0.8, 2.5, 2.5.1, 2.5.2, 2.5.3, 2.5.4, 2.5.5, 2.5.6, 2.5.6.SEC01, 2.5.6.SEC02
Restrictions:	org.springframework

Appendix B

Analysis Results

Box-Plot Representation

For several analyses a box-plot representation is used that depicts the median, minimum, maximum, and average values, as well as the interquartile range that covers 50% of all data points. Figure B.1 illustrates how these data are depicted in this thesis.

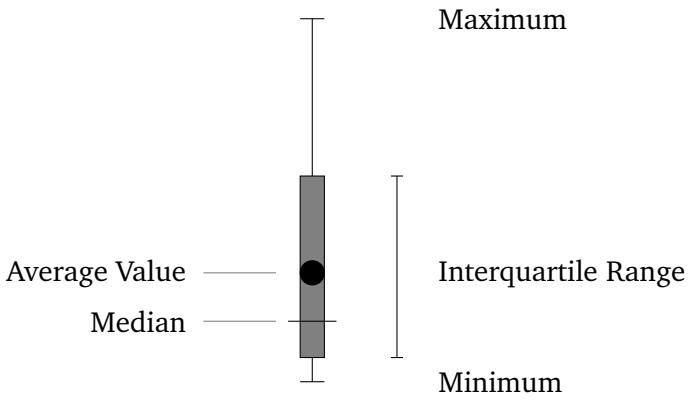


Figure B.1: Explanation of the Box-Plot Representation used in this Thesis

B.1 Consistency

B.1.1 ADN

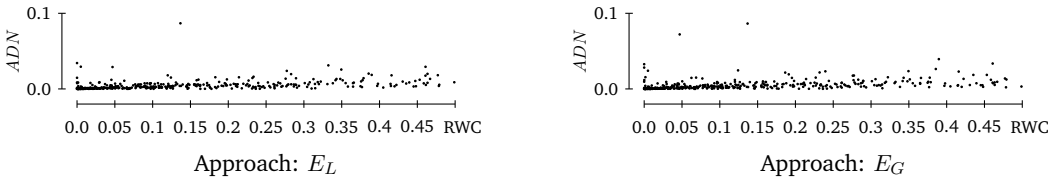


Figure B.2: ADN Plot for E_L and E_G

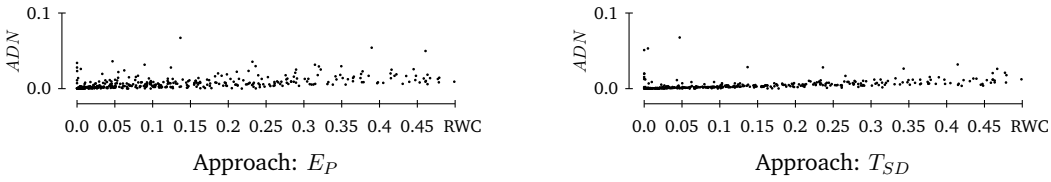


Figure B.3: ADN Plot for E_P and T_{SD}

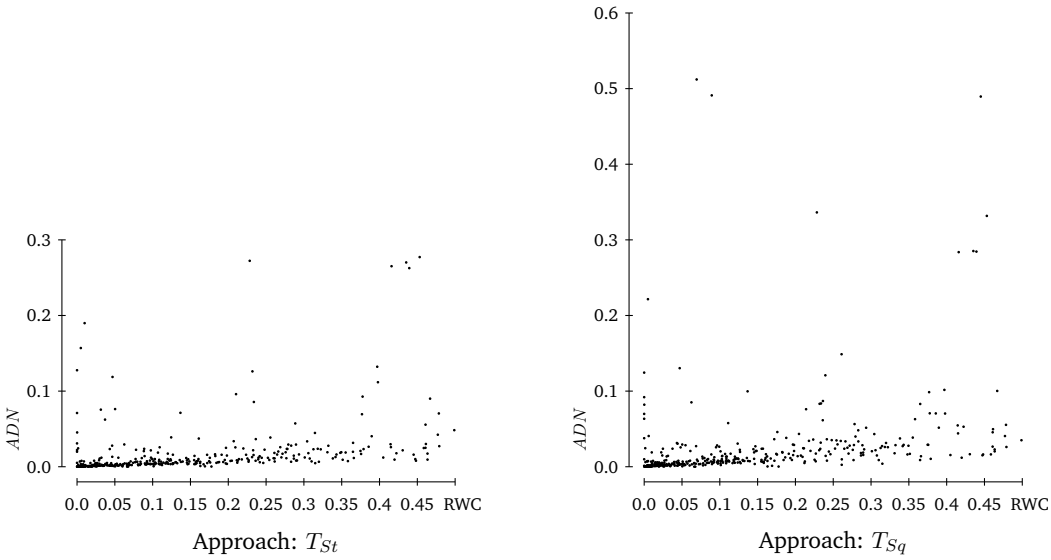


Figure B.4: ADN Plot for T_{St} and T_{Sq}

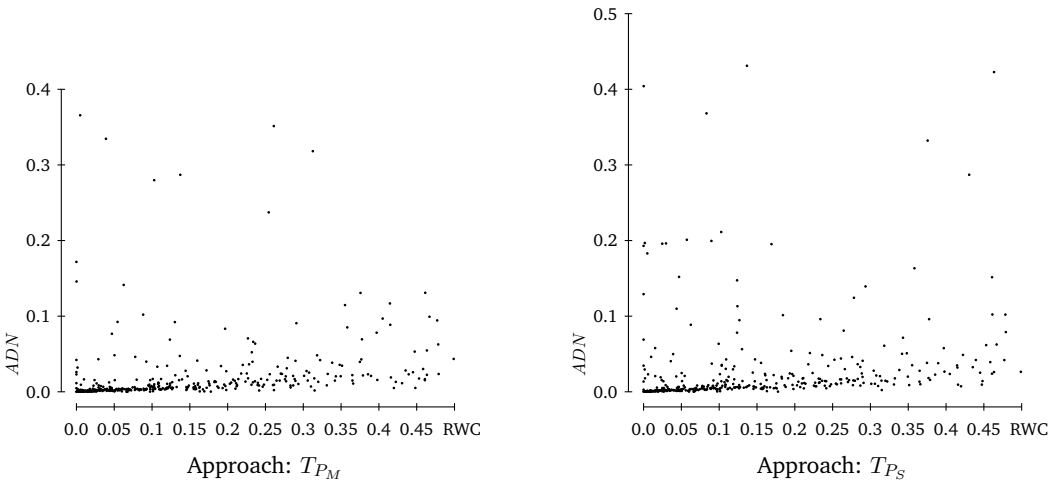


Figure B.5: ADN Plot for T_{P_M} and T_{P_S}

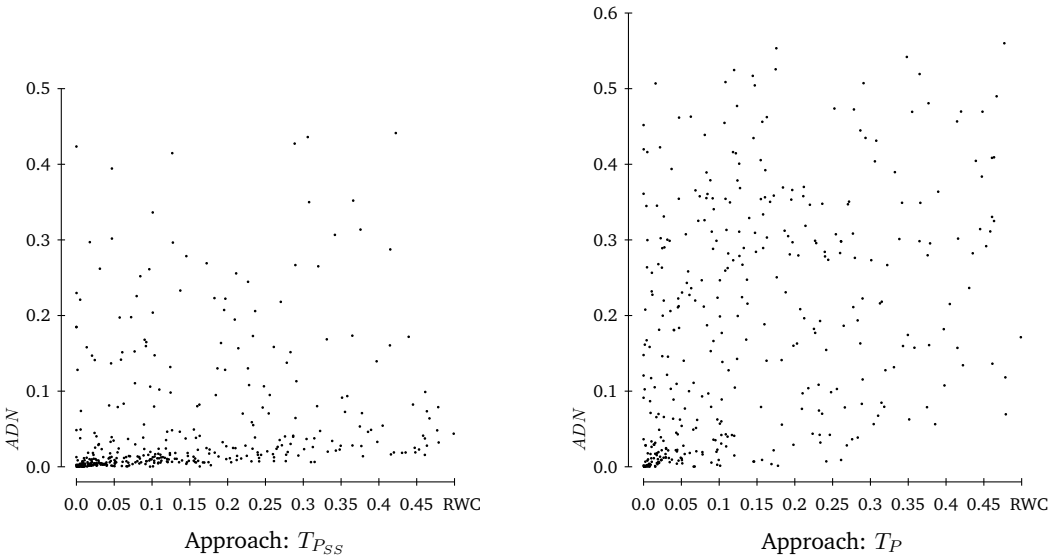
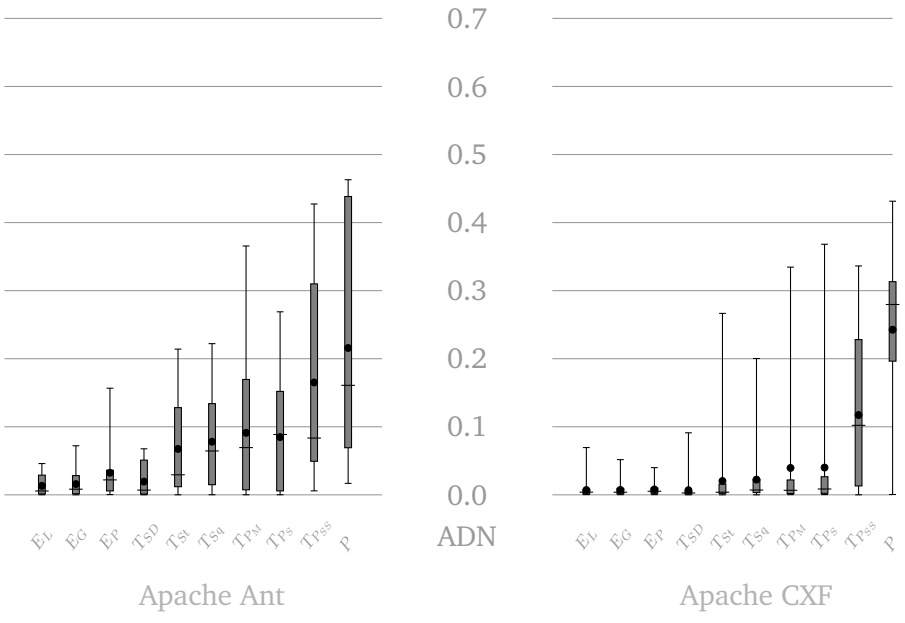
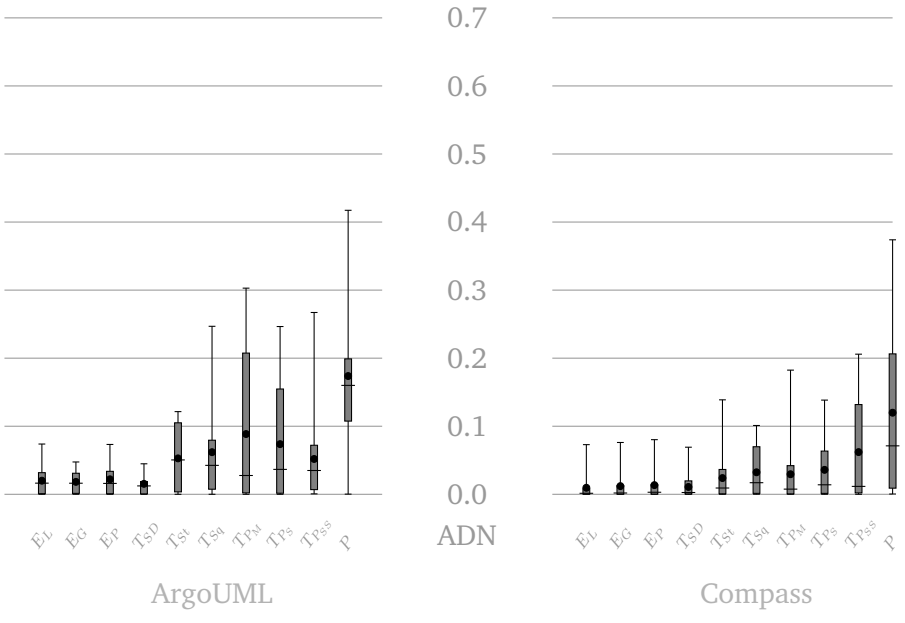
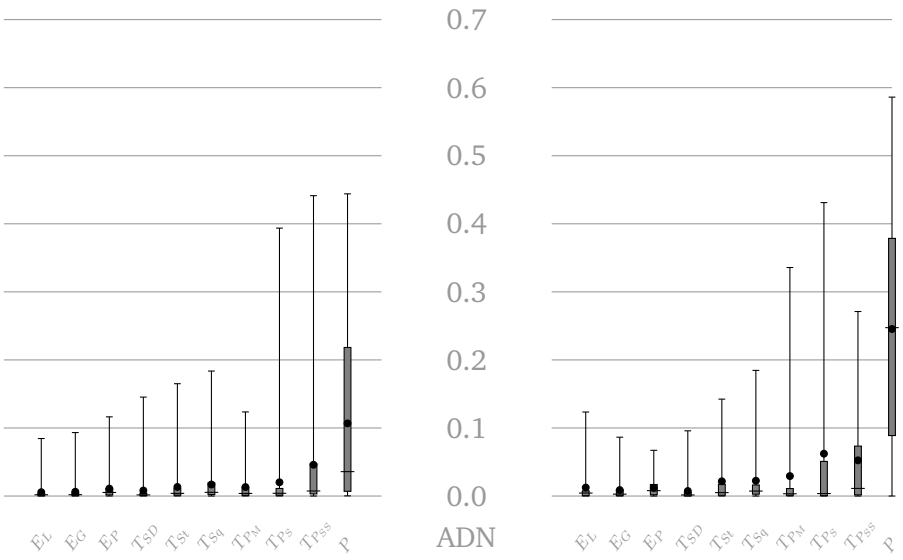
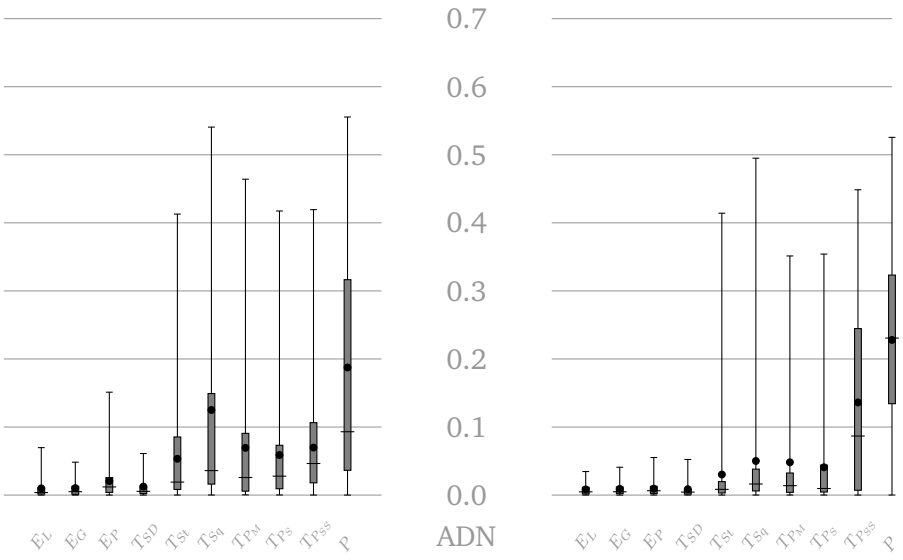


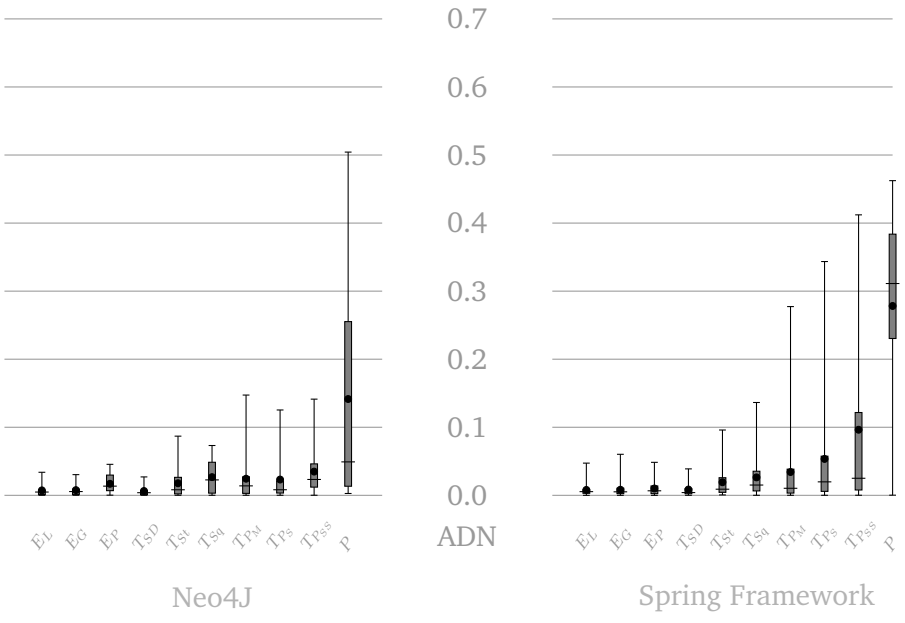
Figure B.6: ADN Plot for $T_{P_{SS}}$ and T_P


0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
ADN
EL EG EP TSD TSI TSA TPM TPS TPSS P
Apache Ant
EL EG EP TSD TSI TSA TPM TPS TPSS P
Apache CXF

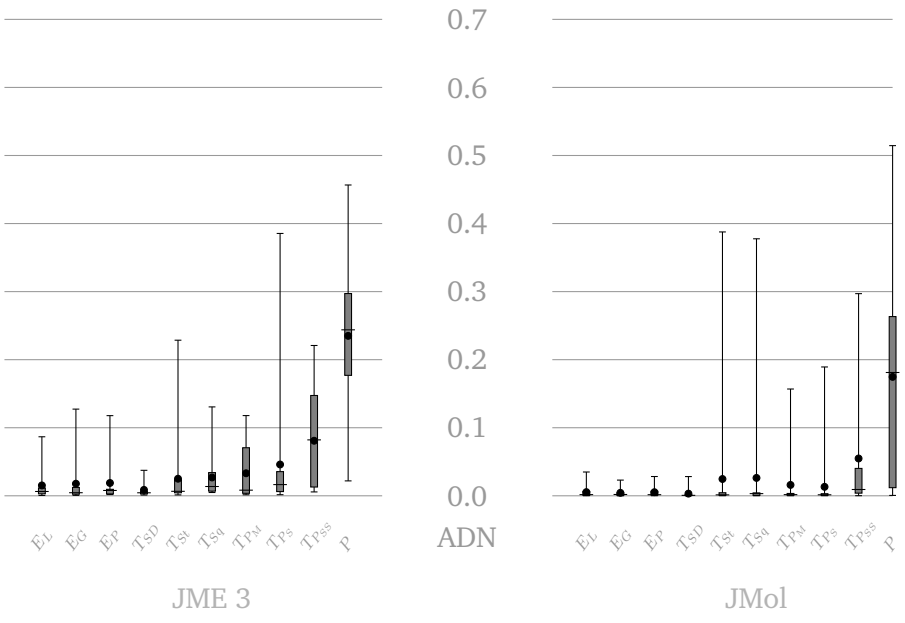

0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
ADN
EL EG EP TSD TSI TSA TPM TPS TPSS P
ArgoUML
EL EG EP TSD TSI TSA TPM TPS TPSS P
Compass


0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
ADN
Datanucleus Core
Hibernate Core

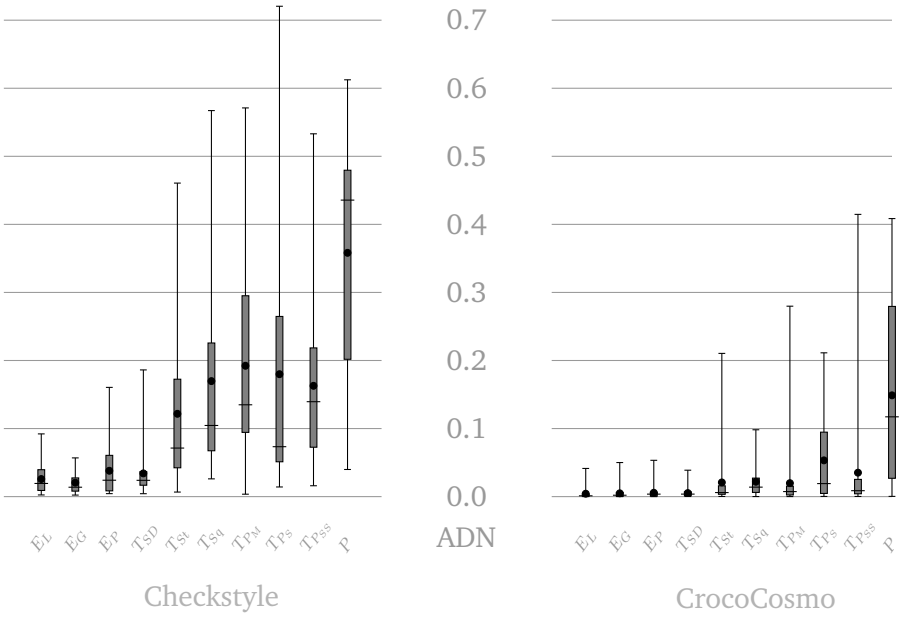

0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
ADN
JFreeChart
Mule Core

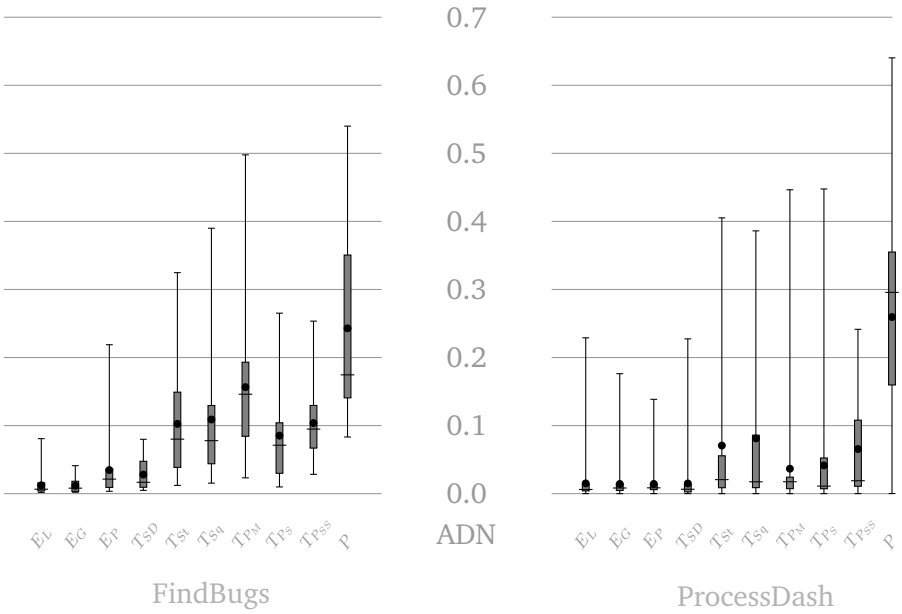


Neo4J Spring Framework



JME 3 JMol


0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
ADN
Checkstyle
CrocoCosmo


0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
ADN
FindBugs
ProcessDash

B.1.2 NNW

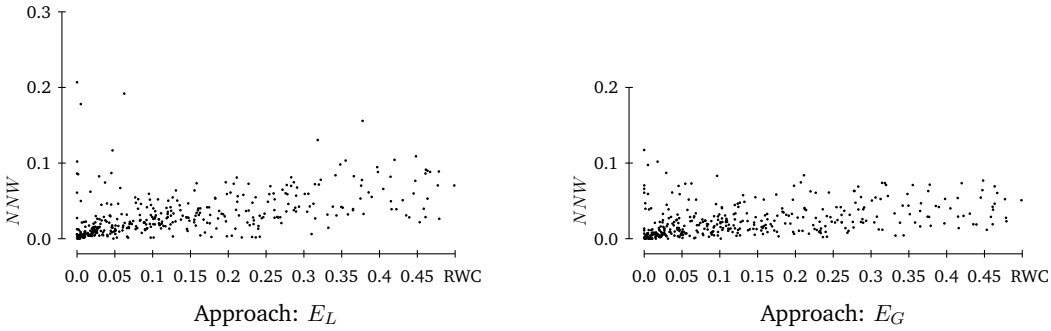


Figure B.7: NNW Plot for E_L and E_G

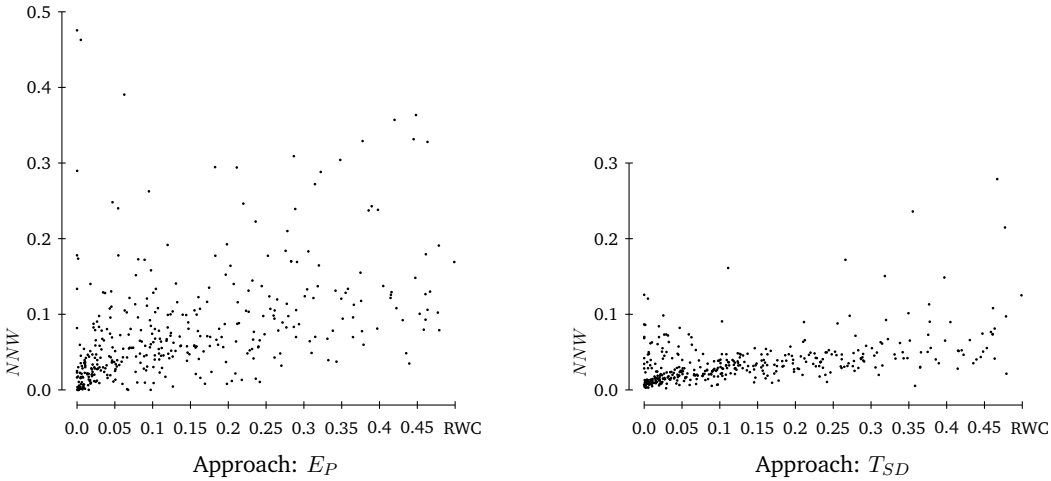


Figure B.8: NNW Plot for E_P and T_{SD}

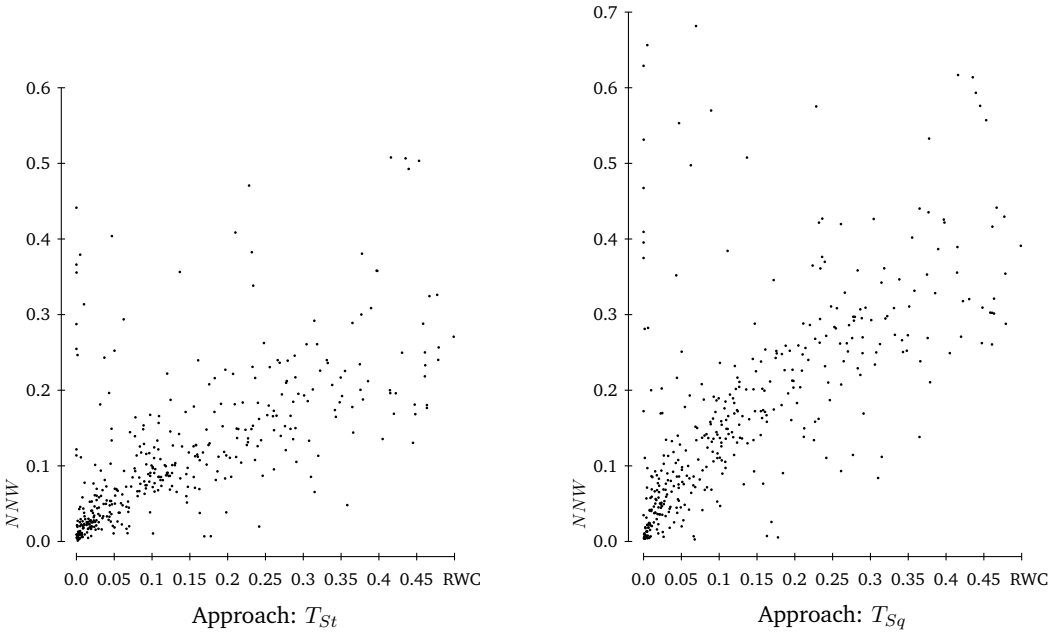


Figure B.9: NNW Plot for T_{St} and T_{Sq}

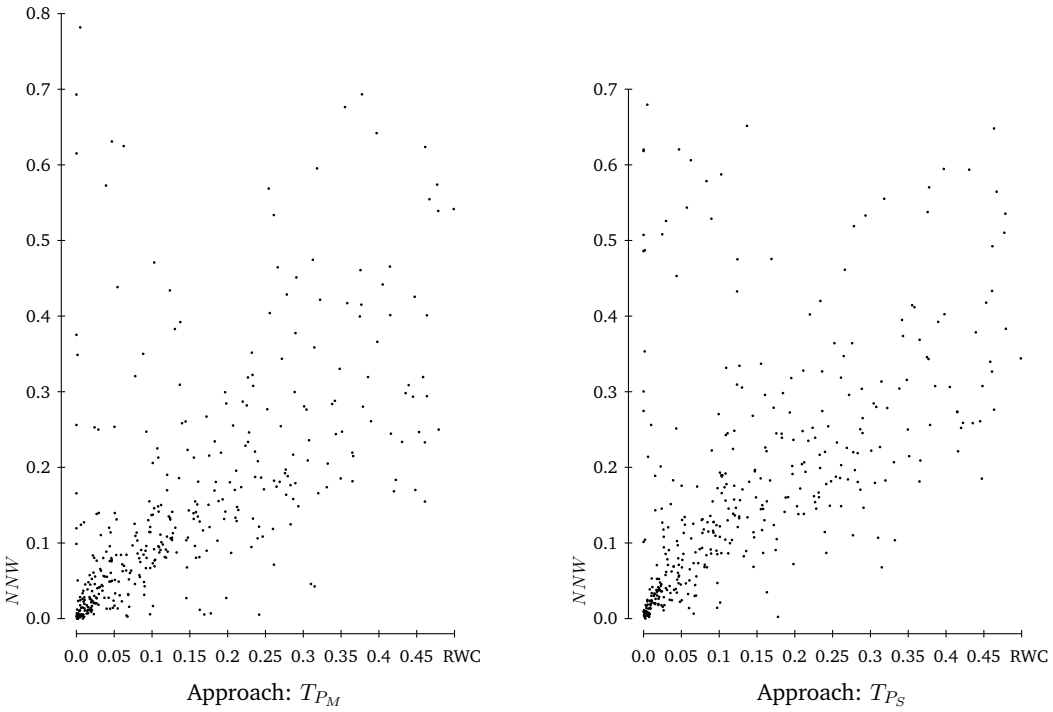


Figure B.10: NNW Plot for T_{P_M} and T_{P_S}

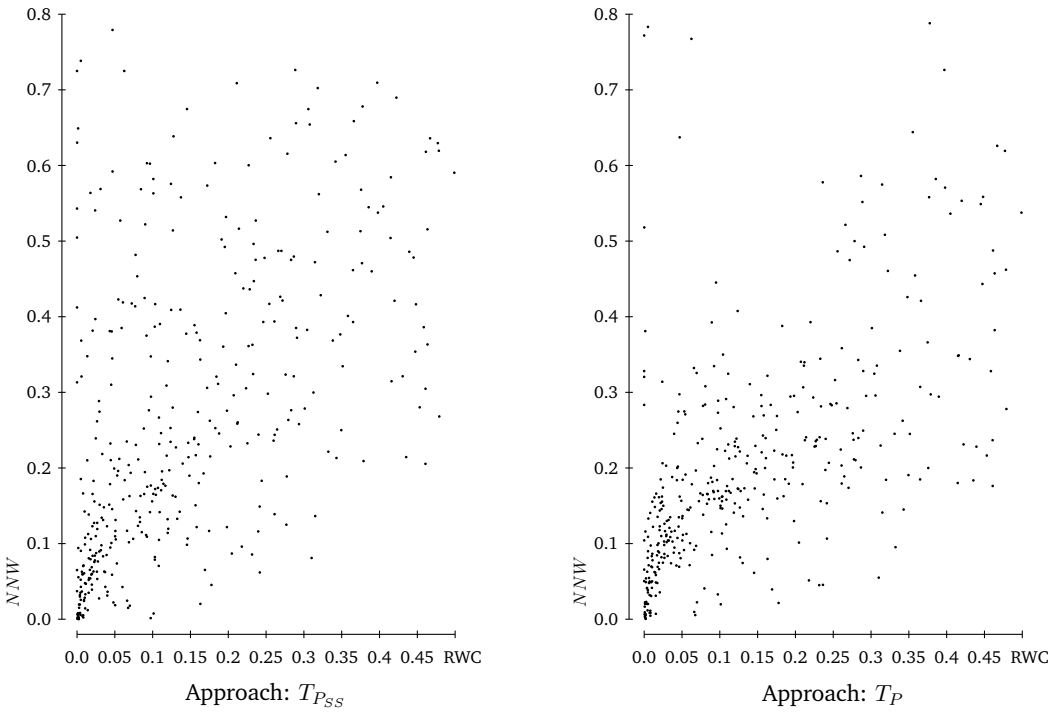
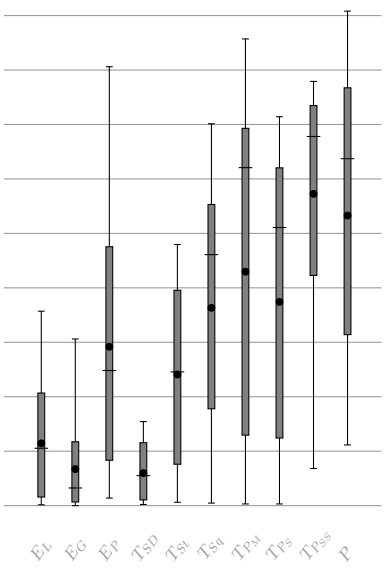
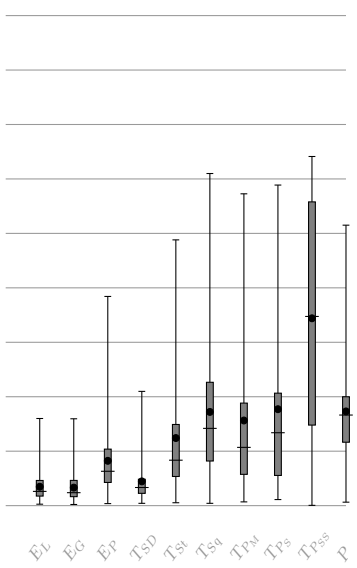


Figure B.11: NNW Plot for $T_{P_{SS}}$ and T_P

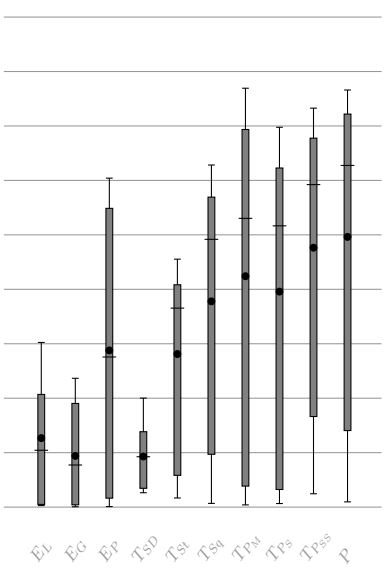


Apache Ant

NNW

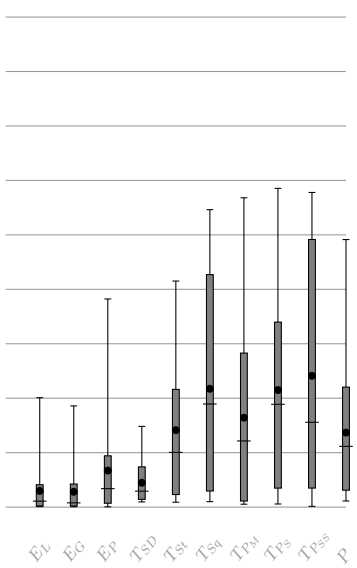


Apache CXF

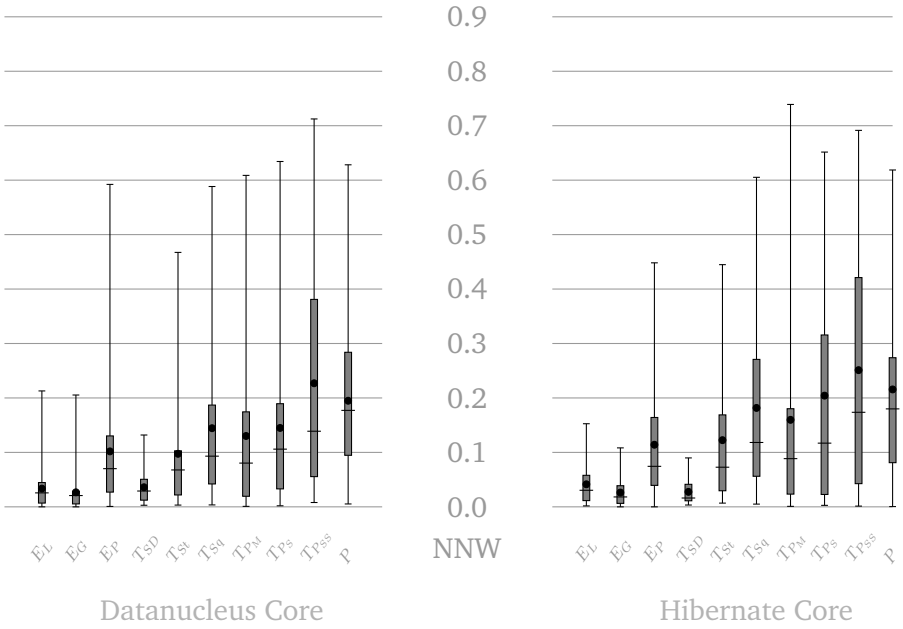


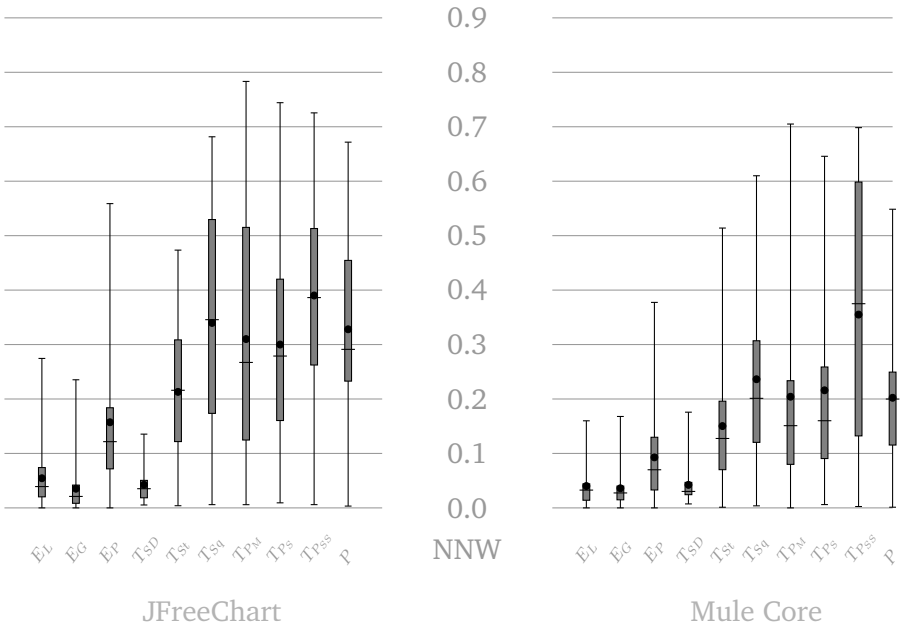
ArgoUML

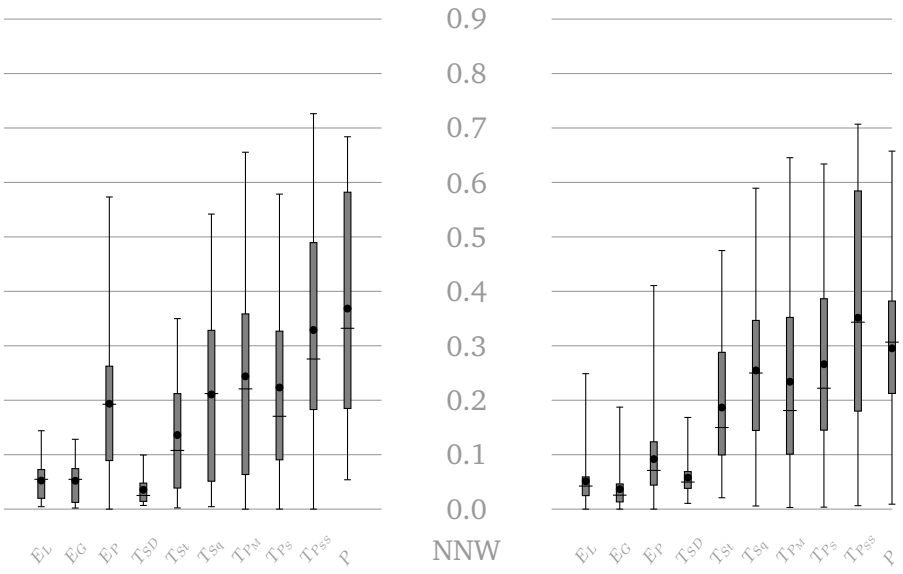
NNW



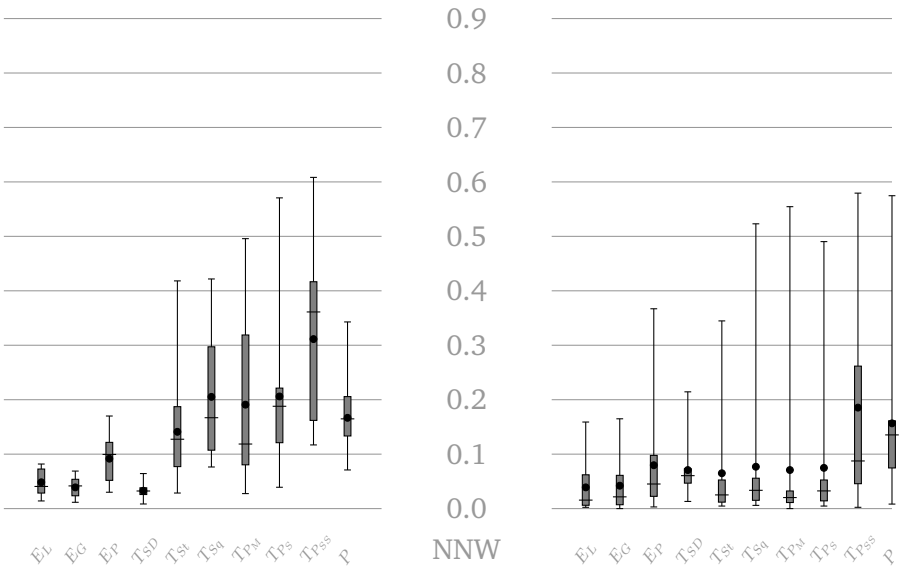
Compass


0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
NNW
EL
EG
EP
TSD
TSI
TSQ
TPM
TPS
TPSS
P
Datanucleus Core
Hibernate Core

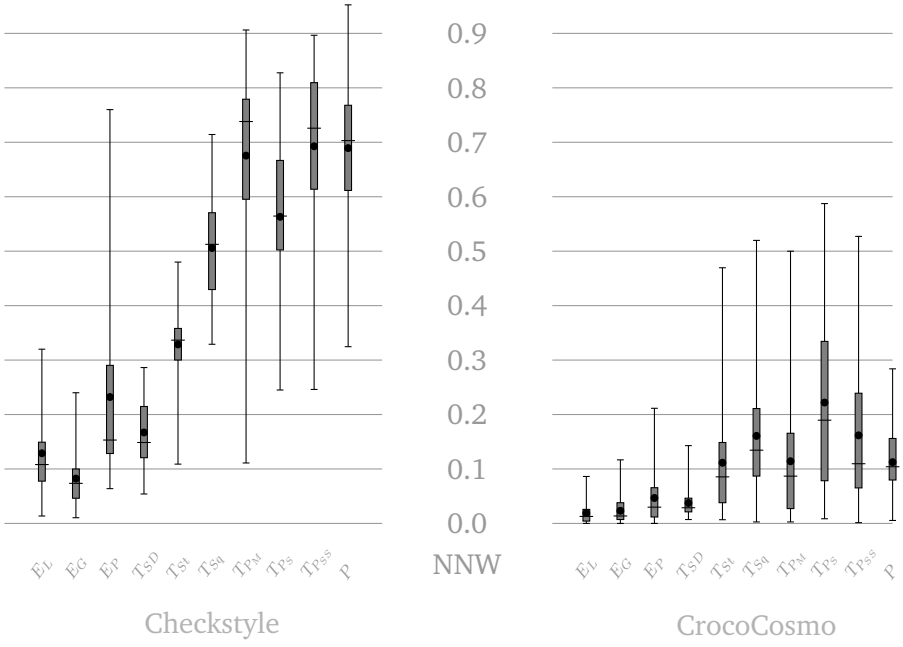

0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
NNW
EL
EG
EP
TSD
TSI
TSQ
TPM
TPS
TPSS
P
JFreeChart
Mule Core

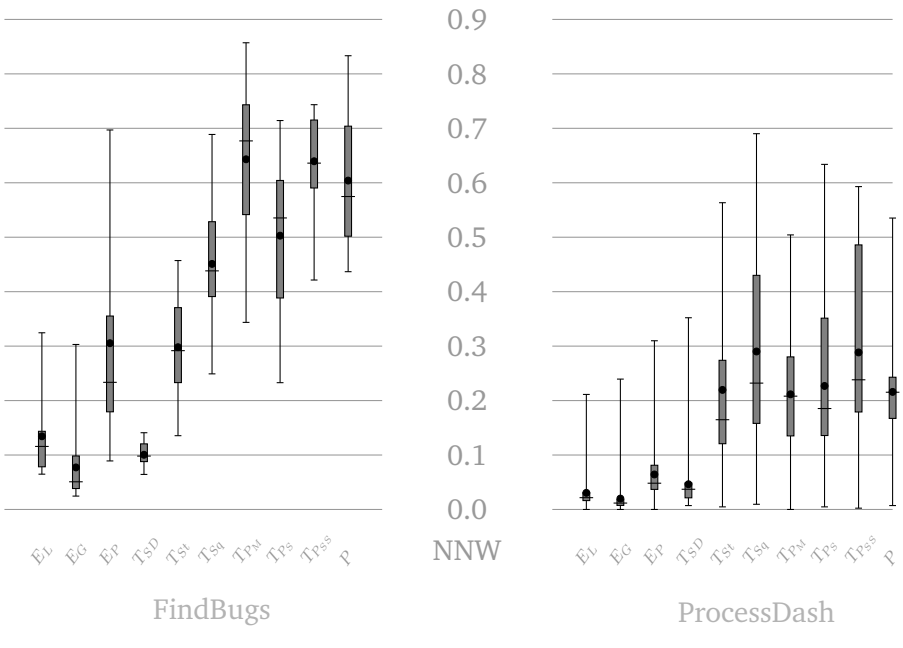


Neo4J Spring Framework



JME 3 JMol


0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
NNW
E_L E_G E_P T_{SD} T_{St} T_{Sa} T_{PM} T_{PS} T_{PSS} P
Checkstyle
CrocoCosmo


0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
NNW
E_L E_G E_P T_{SD} T_{St} T_{Sa} T_{PM} T_{PS} T_{PSS} P
FindBugs
ProcessDash

B.1.3 Ranking

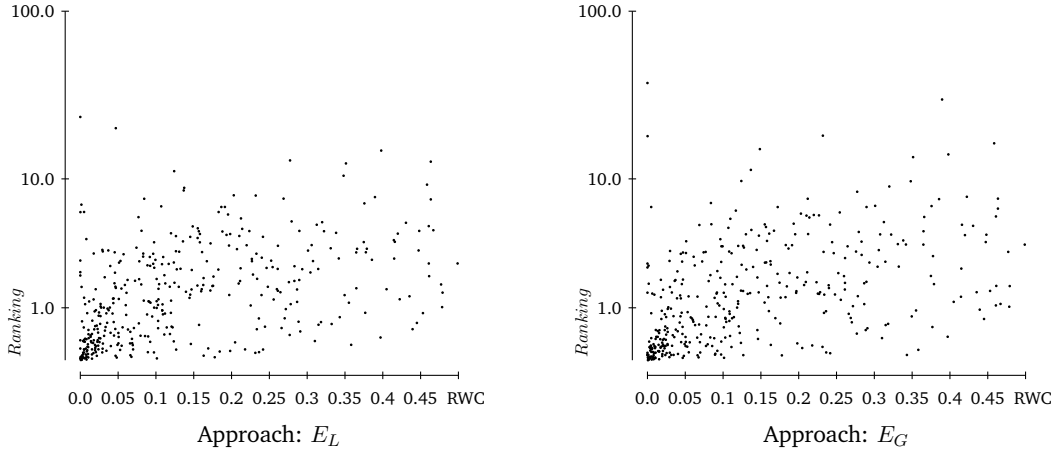


Figure B.12: $Ranking$ Plot for E_L and E_G

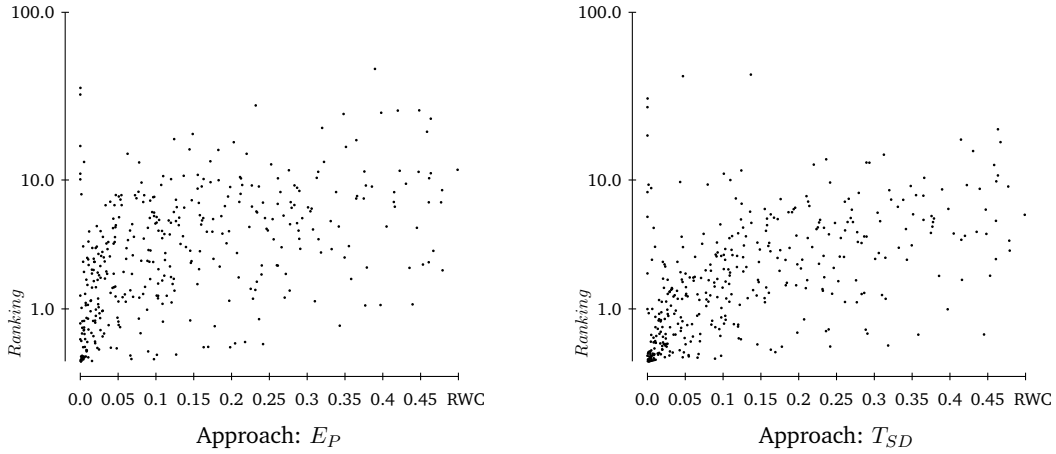


Figure B.13: $Ranking$ Plot for E_P and T_{SD}

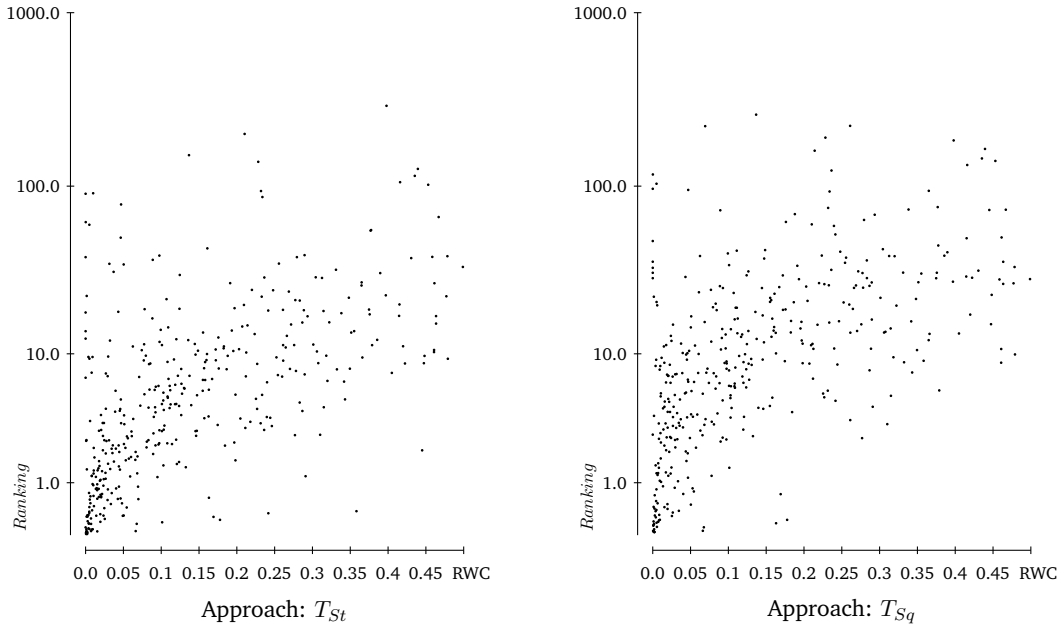


Figure B.14: $Ranking$ Plot for T_{St} and T_{Sq}

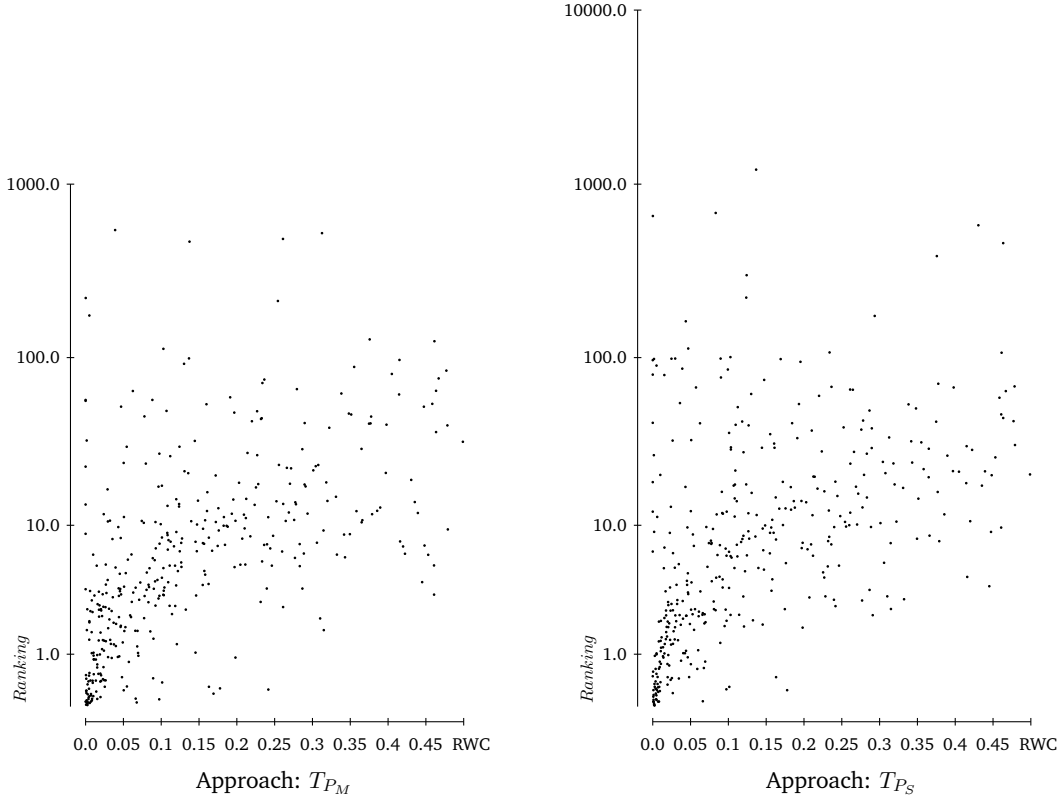


Figure B.15: $Ranking$ Plot for T_{P_M} and T_{P_S}

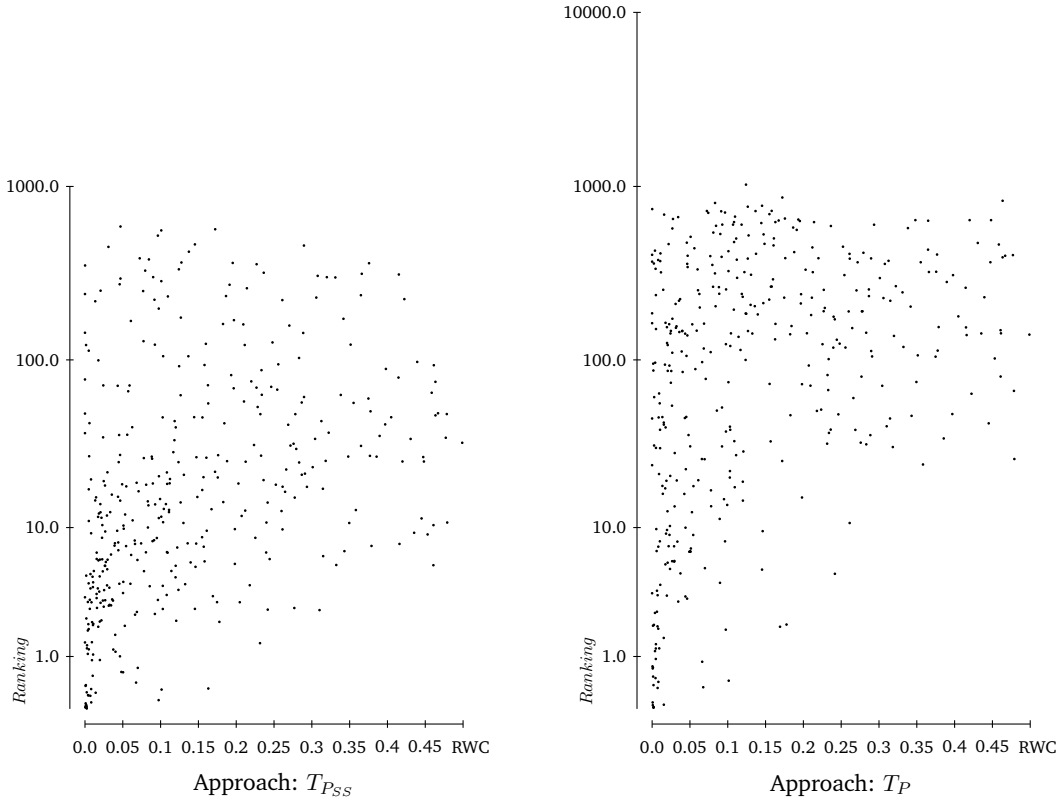
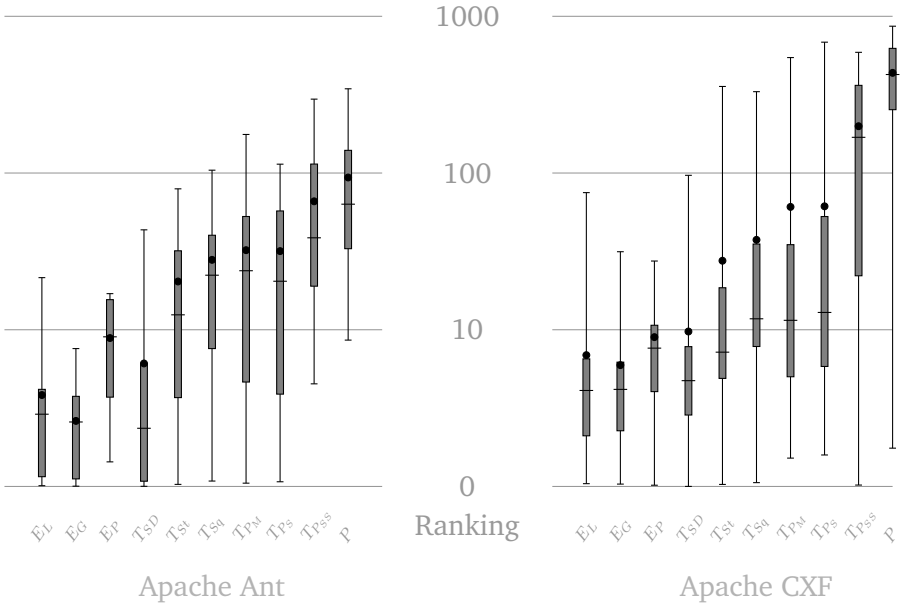
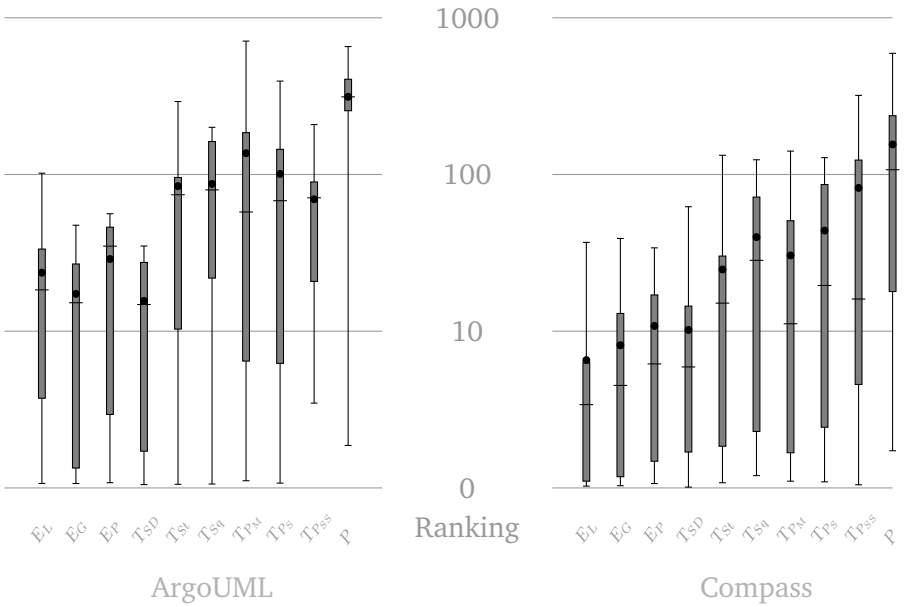
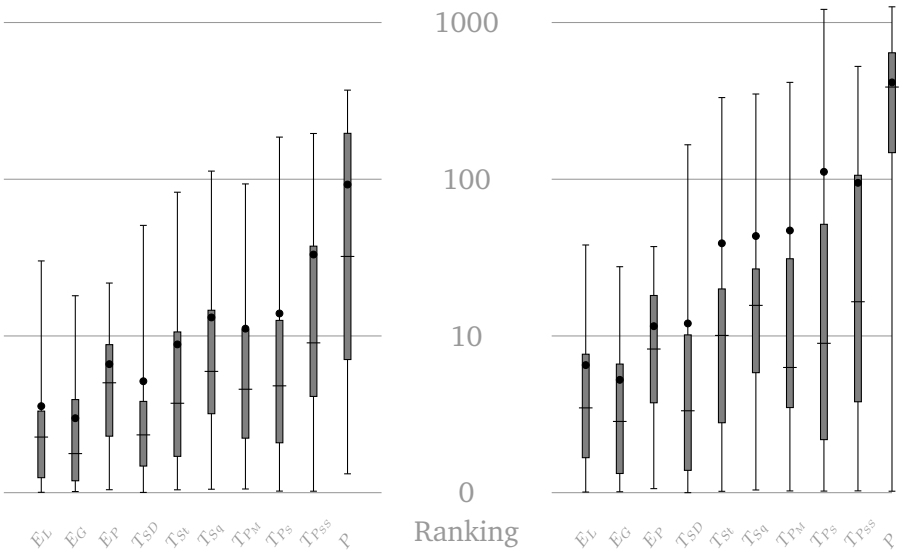


Figure B.16: $Ranking$ Plot for $T_{P_{SS}}$ and T_P

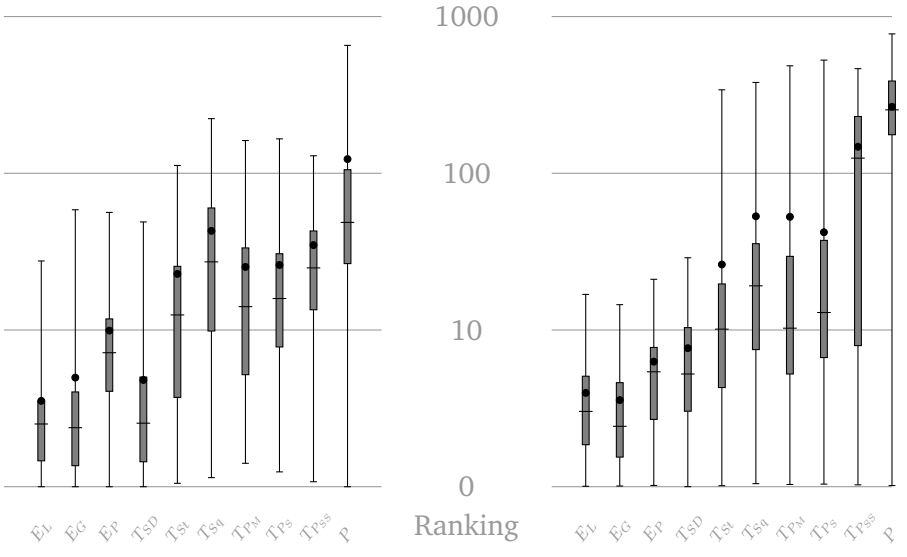

1000
100
10
0
Ranking
E_L E_G E_P T_SD T_St T_Sa T_PM T_Ps T_PSS P
Apache Ant
Apache CXF


1000
100
10
0
Ranking
E_L E_G E_P T_SD T_St T_Sa T_PM T_Ps T_PSS P
ArgoUML
Compass



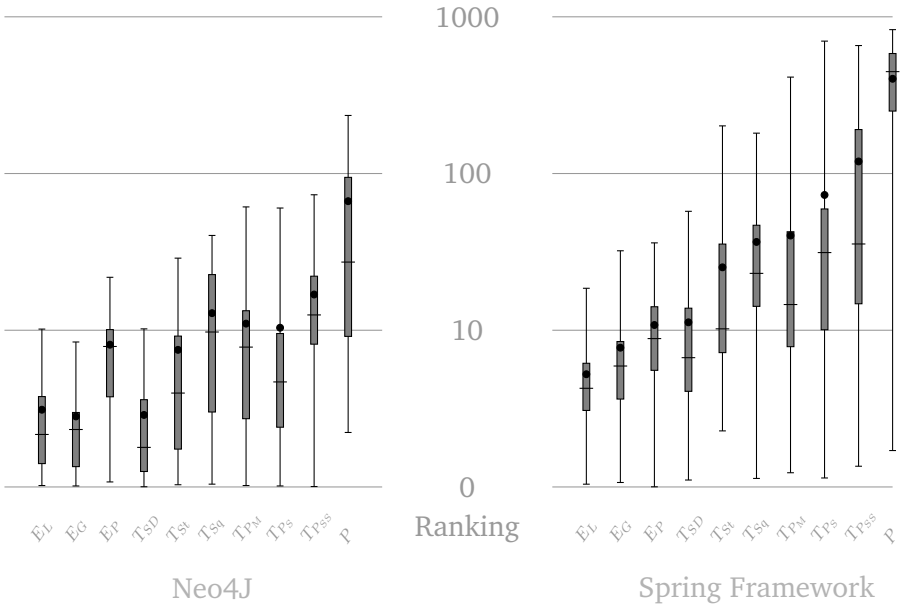
Datanucleus Core

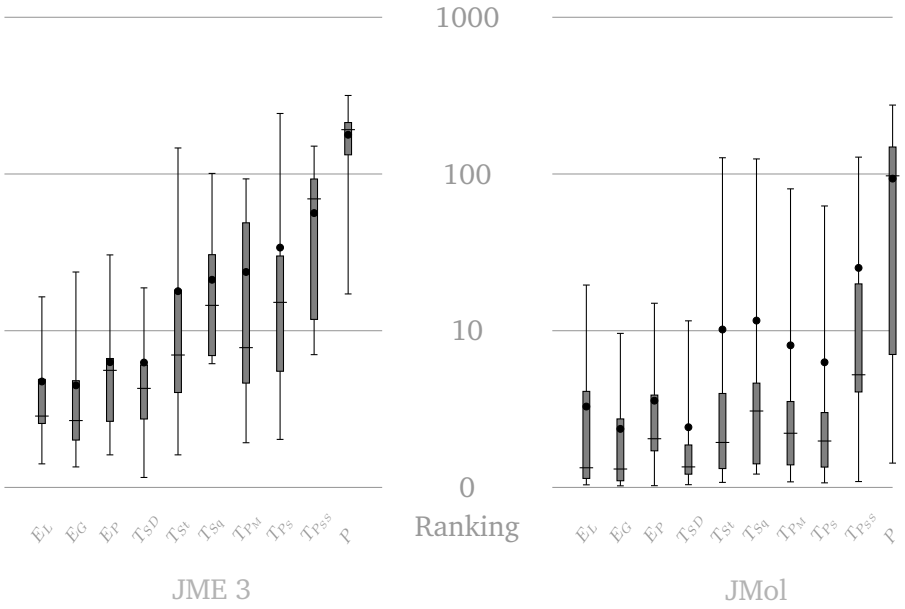
Hibernate Core

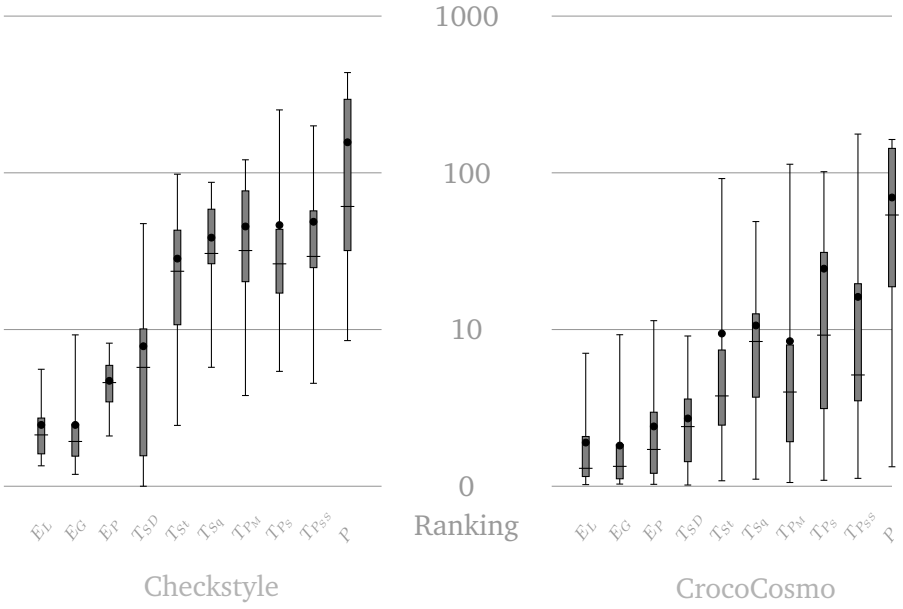


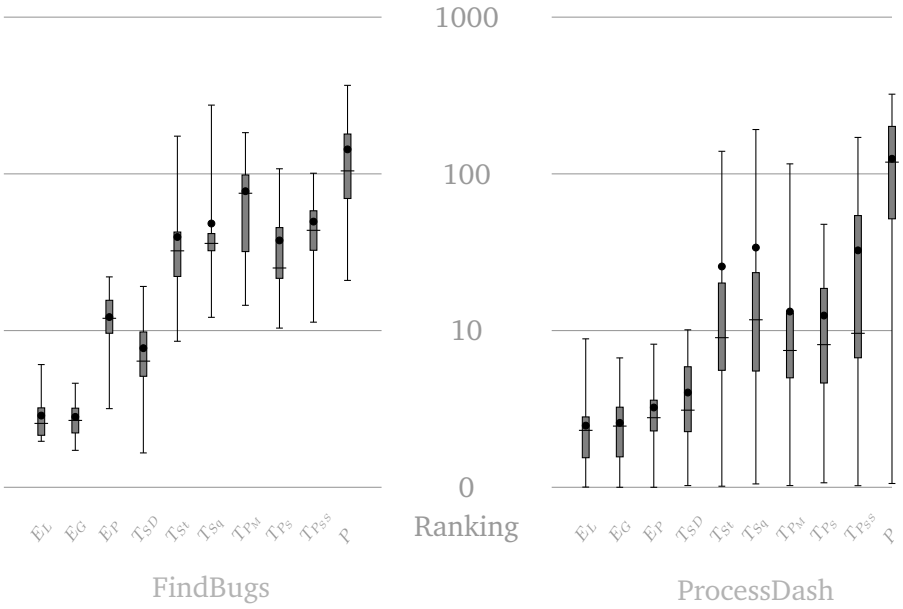
JFreeChart

Mule Core


1000
100
10
0
Ranking
EL
EG
EP
TSD
TSI
TSA
TPM
TPS
TPSS
P
Neo4J
Spring Framework


1000
100
10
0
Ranking
EL
EG
EP
TSD
TSI
TSA
TPM
TPS
TPSS
P
JME 3
JMol


1000
100
10
0
Ranking
EL
EG
EP
TSD
TSI
TSQ
TPM
TPS
TPSS
P
Checkstyle
CrocoCosmo


1000
100
10
0
Ranking
EL
EG
EP
TSD
TSI
TSQ
TPM
TPS
TPSS
P
FindBugs
ProcessDash

B.2 Compactness

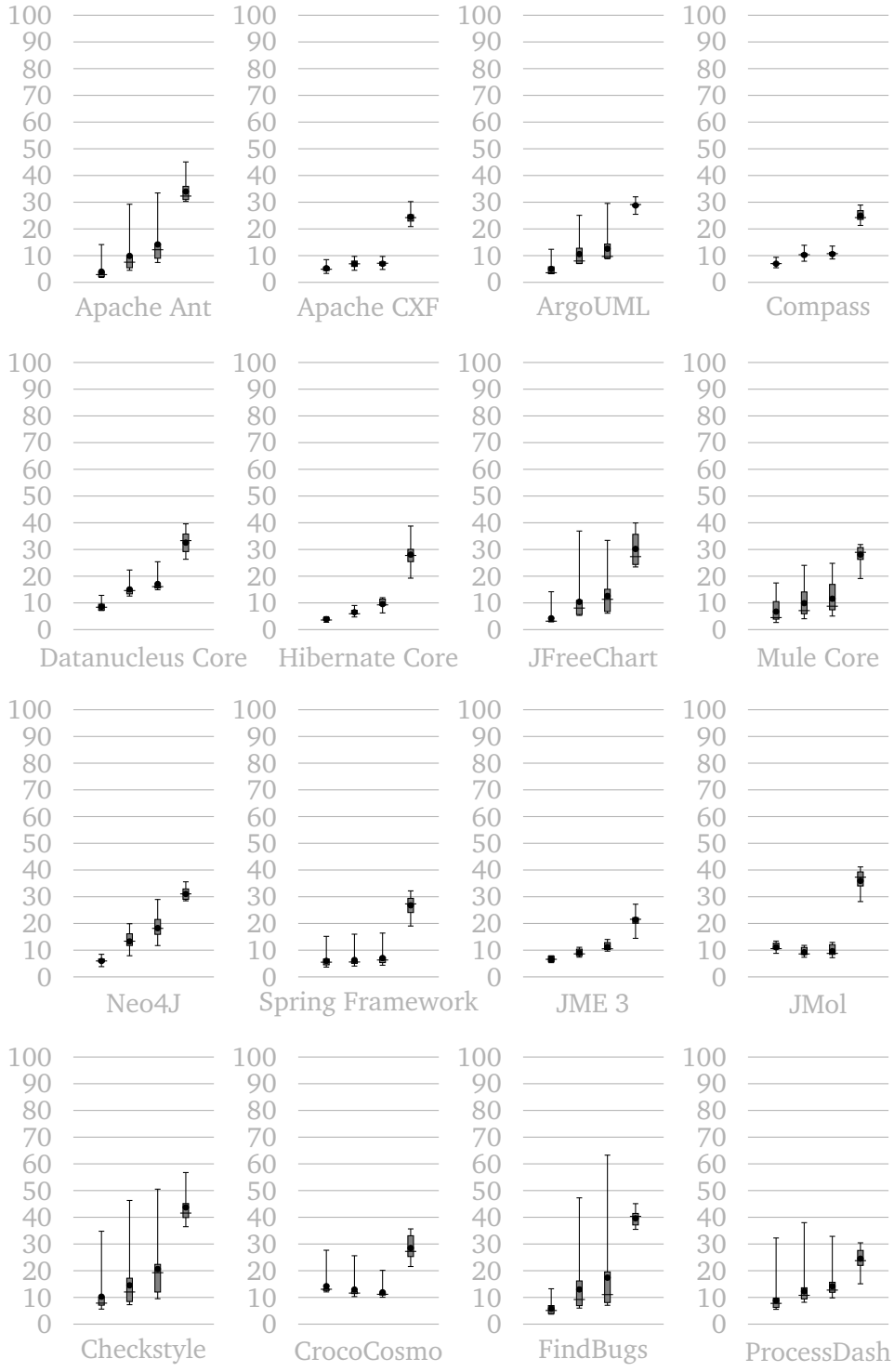


Figure B.17: SEC Compactness for E_L, E_G, E_P, P

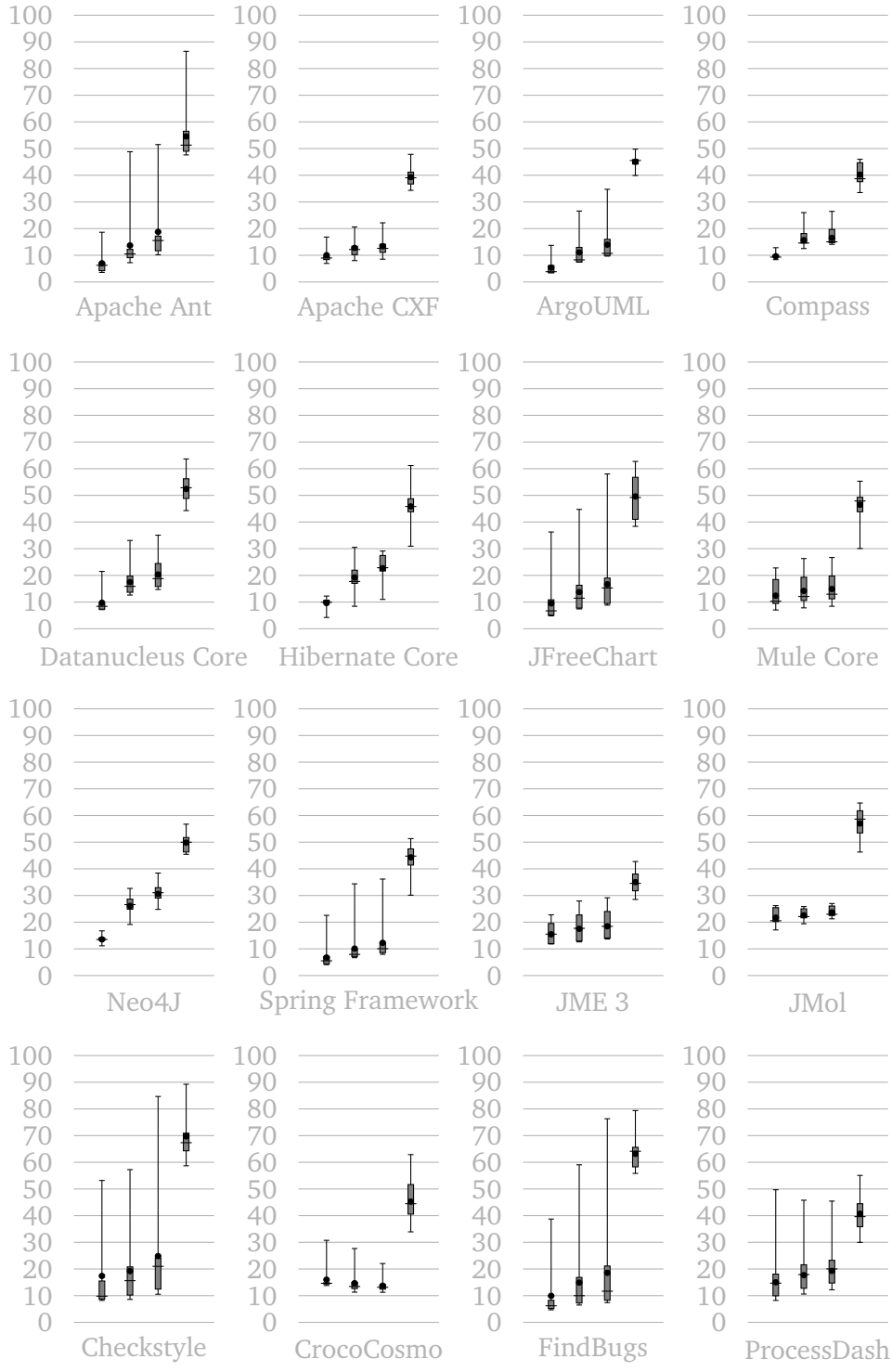


Figure B.18: SER Compactness for E_L, E_G, E_P, P

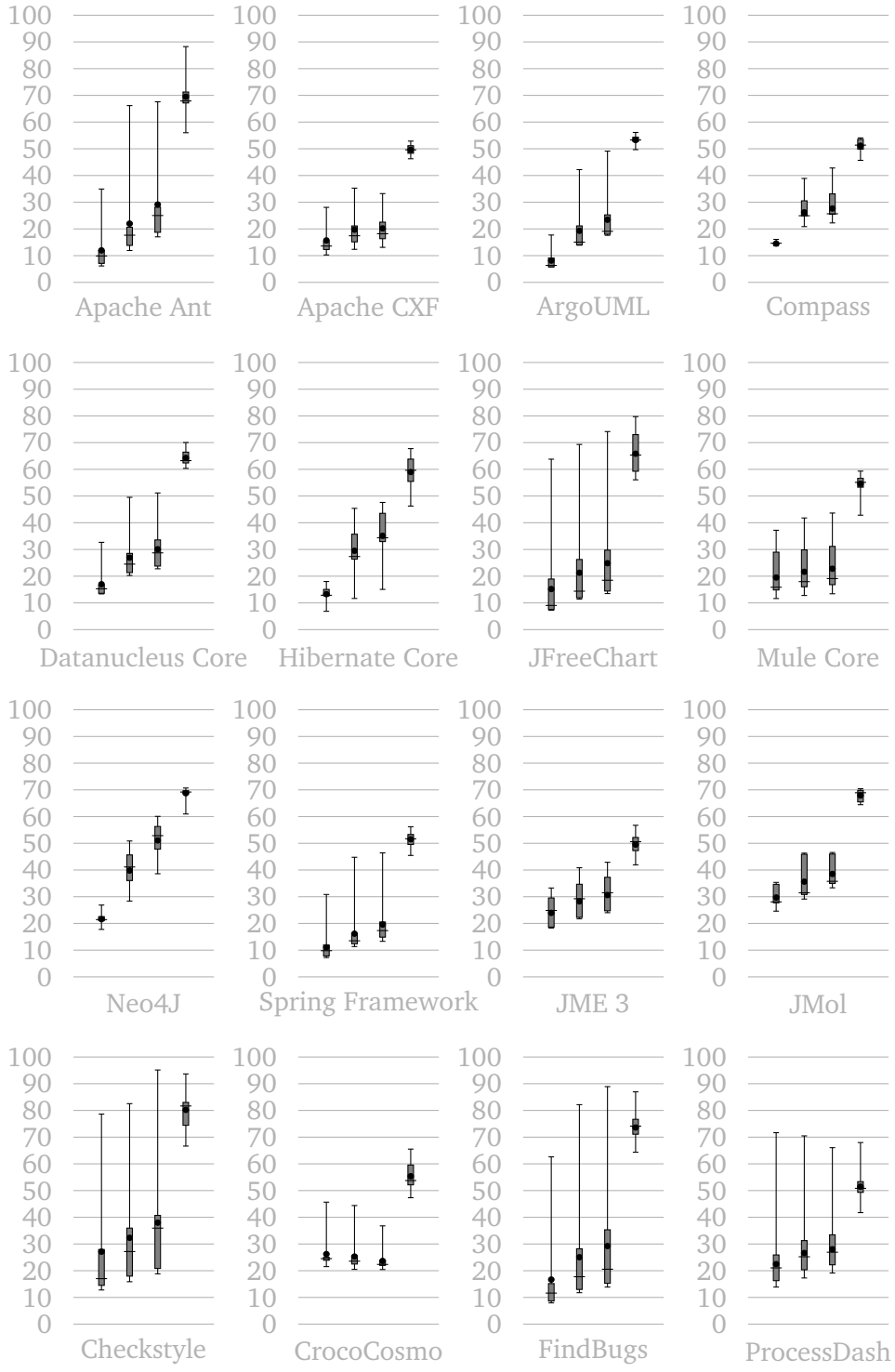
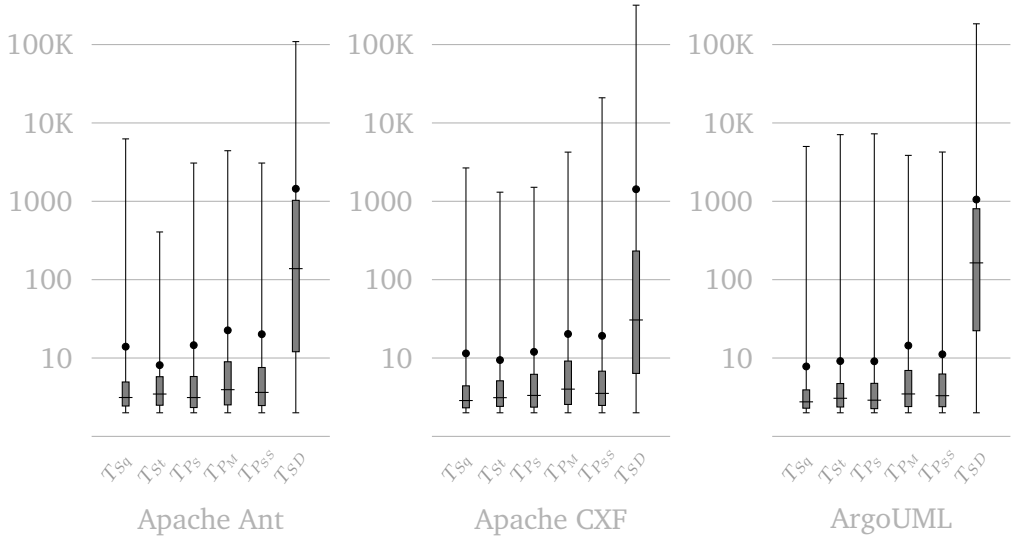
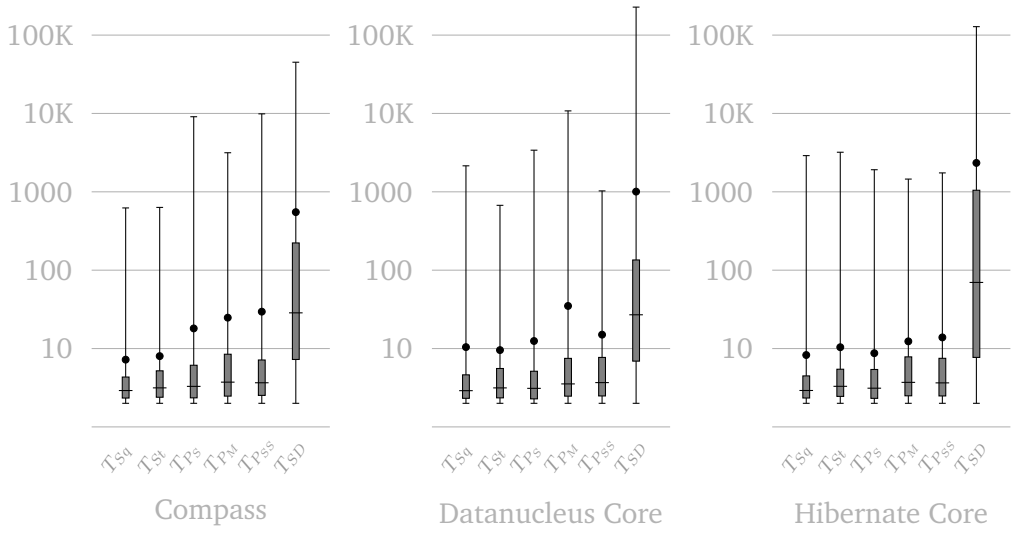
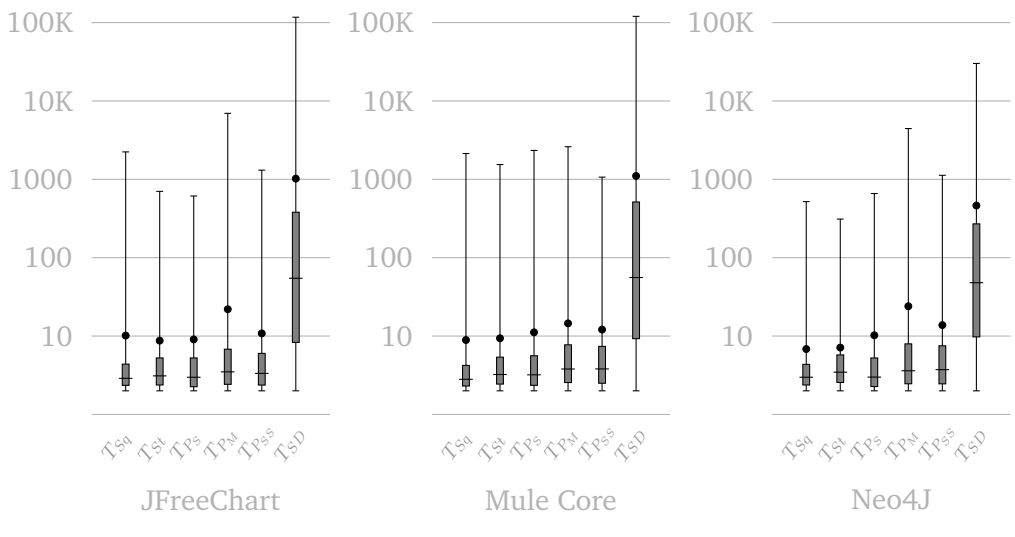


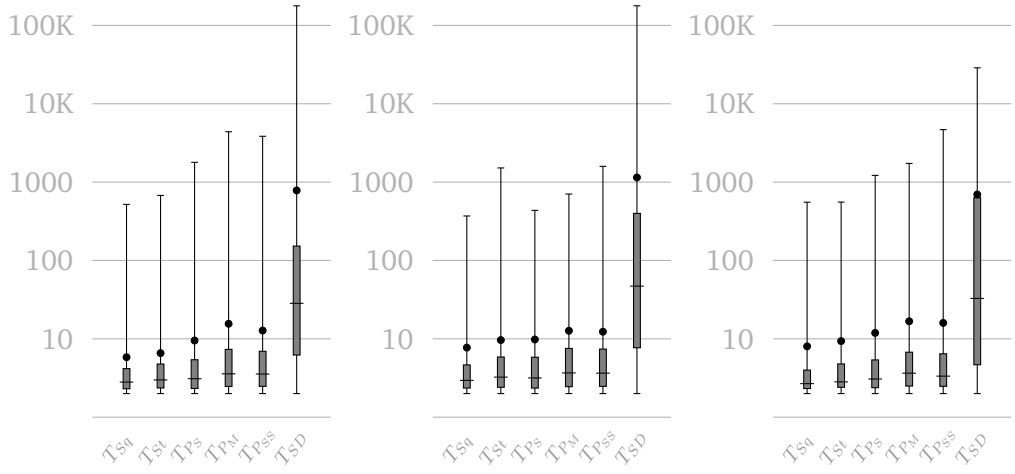
Figure B.19: SEP Compactness for E_L, E_G, E_P, P

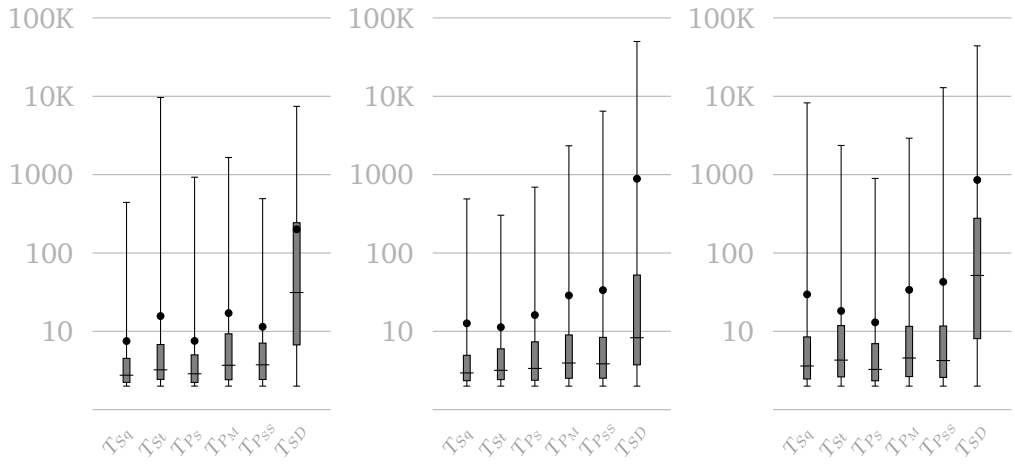
B.3 Aspect Ratios

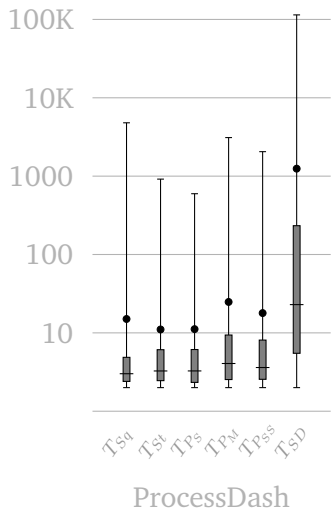

100K
10K
1000
100
10
T_{SQ} T_{ST} T_{PS} T_{PM} T_{PSS} T_{SD}
Apache Ant
Apache CXF
ArgoUML


100K
10K
1000
100
10
T_{SQ} T_{ST} T_{PS} T_{PM} T_{PSS} T_{SD}
Compass
Datanucleus Core
Hibernate Core


100K
10K
1000
100
10
T_{SQ} T_{ST} T_{PS} T_{PM} T_{PSS} T_{SD}
JFreeChart
Mule Core
Neo4J


100K
10K
1000
100
10
T_{SQ} T_{ST} T_{PS} T_{PM} T_{PSS} T_{SD}
Spring Framework
JME 3
JMol


100K
10K
1000
100
10
T_{SQ} T_{ST} T_{PS} T_{PM} T_{PSS} T_{SD}
Checkstyle
CrocoCosmo
FindBugs


100K
10K
1000
100
10
T_{SQ} T_{ST} T_{PS} T_{PM} T_{PSS} T_{SD}
ProcessDash

Publications

Parts of this thesis have previously been published in the following publications related to this work:

- Frank Steinbrückner, Claus Lewerentz: Understanding Software Evolution with Software Cities. Accepted for publication. Information Visualization Journal. Sage Publications. 2012.
- Frank Steinbrückner, Claus Lewerentz: Representing Development History in Software Cities, Proceedings of the 5th International Symposium on Software Visualization, ISBN 978-1-4503-0028-5, pp. 193–202, 2010, Salt Lake City, Utah, USA. ACM, New York, NY, USA.
- Frank Steinbrückner: Coherent Software Cities. Supporting Comprehension of Evolving Software Systems, Proceedings of IEEE International Conference on Software Maintenance, Timisoara, Romania. 2010. ISBN 978-1-4244-8630-4, IEEE Computer Society, Los Alamitos, CA, USA.
- Claus Lewerentz; Frank Steinbrückner: SoftUrbs: Visualizing Software Systems as Urban Structures, Computer Science Reports 02/2009, Brandenburg University of Technology Cottbus, 2009.
- Marcel Bennicke, Frank Steinbrückner, Mathias Radicke, Jan-Peter Richter: Das sd&m Software Cockpit: Architektur und Erfahrungen. In: Rainer Koschke, Otthein Herzog, Karl-Heinz Rödiger, Marc Ronthaler: GI Jahrestagung (2). pp. 254-260, 2007. ISBN 978-3-88579-204-8.

PUBLICATIONS PRIOR TO THIS WORK

- Claus Lewerentz, Frank Simon, Frank Steinbrückner, CrocoCosmos, In: Petra Mutzel, Michael Jünger, Sebastian Leipert (Eds.): Graph Drawing. 9th International Symposium, GD 2001. Vienna, Austria, September 2001, Revised Papers. LNCS 2265, pages 446 - 447, Springer-Verlag, 2002, ISBN 3-540-43309-0
- Claus Lewerentz, Frank Simon, Frank Steinbrückner, Holger Breitling, Carola Lilienthal, Martin Lippert: External Validation of a Metric-based Quality Assessment of the JWAM Framework. In: Dumke/Rombach: Software-Messung und Bewertung, Deutscher Universitätsverlag, Wiesbaden, Germany, 2002, pp. 32-49
- Frank Simon, Frank Steinbrückner, Claus Lewerentz: Metrics Based Refactoring, Proceedings of the 5th European Conference on Software Maintenance and Reengineering (CSMR 2001), pp. 30-38, IEEE Computer Society Press, 2001

- Frank Simon, Frank Steinbrückner, Claus Lewerentz: Anpaßbare, explorierbare virtuelle Informationsräume zur Qualitätsbewertung großer Software-Systeme, presented on 3rd Workshop on Reengineering in Bad Honnef, 2001
- Frank Simon, Frank Steinbrückner, Claus Lewerentz: 3D-Spring Embedder for Complete Graphs. Technical Report 11/00, Computer Science Reports, Brandenburg University of Technology Cottbus, October 2000
- Frank Simon, Claus Lewerentz, Frank Steinbrückner: Multidimensionale Mess- und Strukturbasierte Softwarevisualisierung. Proceedings of 2nd Workshop Reengineering in Bad Honnef, Mai 2000